PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 3 : G06F 7/00, 9/00, 15/00

A1

(11) International Publication Number: WO 86/ 00434

(43) International Publication Date: 16 January 1986 (16.01.86)

(21) International Application Number: PCT/US85/00670

(22) International Filing Date: 16 April 1985 (16.04.85)

(31) Priority Application Number: 624,864

(32) Priority Date: 27 June 1984 (27.06.84)

(33) Priority Country: US

(71) Applicant: MOTOROLA, INC. [US/US]; 1303 E. Algonquin Road, Schaumburg, IL 60196 (US).

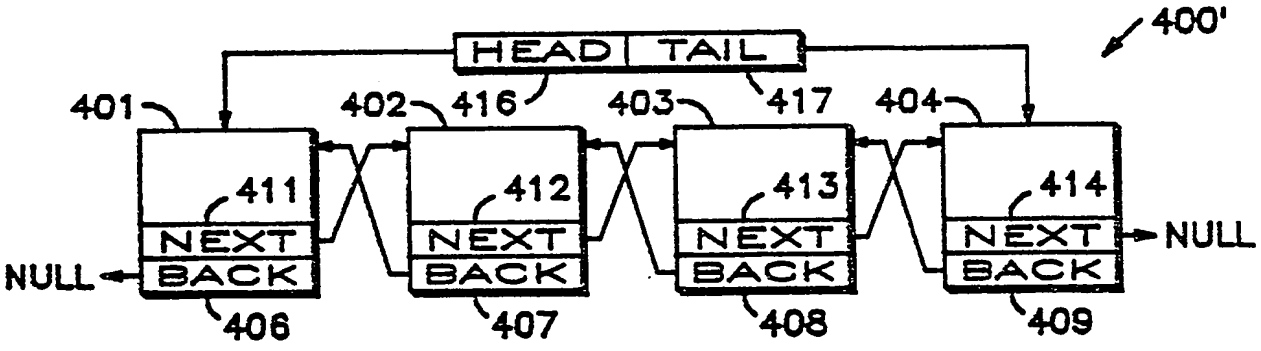
(72) Inventors: MacGREGOR, Douglas ; 3705 Tarragona Lane, Austin, TX 78727 (US). MOTHERSOLE, David, Scott ; 903 Huntridge Drive, Austin, TX 78758 (US). ZOLNOWSKY, John ; 9 Homer Lane, Menlo Park, CA 94025 (US).

(74) Agents: GILLMAN, James, W. et al.; Motorola, Inc., Patent Department - Suite 300K, 4250 E. Camelback Road, Phoenix, AZ 85018 (US).

(81) Designated States: DE (European patent), FR (European patent), GB (European patent), IT (European patent), JP, KR, NL (European patent).

Published
With international search report.

(54) Title: METHOD AND APPARATUS FOR A COMPARE AND SWAP INSTRUCTION



(57) Abstract

In a data processing system (10) having linked lists (400, 400', 500, 500') it is useful to be able to add and delete items from such lists (400, 400', 500, 500') while maintaining the integrity of the linked nature of such lists (400, 400', 500, 500'). A new compare and swap instruction provides for effectively simultaneously swapping 2 values which is useful for safely adding and deleting items (404, 512) from linked lists (400, 400', 500, 500'). Prior to the instruction the status of the two values are read at the locations to be swapped. During the instruction these locations are checked again to ensure that no change has occurred at these locations before the instruction performs the swap of the two new values. The instruction then performs the proposed 2 value swap but only if no change has occurred at these two locations where the swap is to be performed.

FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AT	Austria
AU	Australia
BB	Barbados
BE	Belgium
BG	Bulgaria
BR	Brazil
CF	Central African Republic
CG	Congo
CH	Switzerland
CM	Cameroon
DE	Germany, Federal Republic of
DK	Denmark
FI	Finland
FR	France
GA	Gabon
GB	United Kingdom
HU	Hungary
IT	Italy
JP	Japan
KP	Democratic People's Republic of Korea
KR	Republic of Korea
LI	Liechtenstein
LK	Sri Lanka
LU	Luxembourg
MC	Monaco
MG	Madagascar
ML	Mali
MR	Mauritania
MW	Malawi
NL	Netherlands
NO	Norway
RO	Romania
SD	Sudan
SE	Sweden
SN	Senegal
SU	Soviet Union
TD	Chad
TG	Togo
US	United States of America

METHOD AND APPARATUS FOR A COMPARE AND SWAP INSTRUCTION

Field of the Invention

The subject application relates to instructions for data processors, and more particularly, to instructions for manipulating items in a list in a data processor.

Background of the Invention

One aspect typical of a data processing system is operating with lists. Each list is comprised of a number of items. Each item is something to be performed or used by the data processing system. Items are typically processes, data, programs, or subroutines. One way of keeping track of a list is to have a head pointer and a tail pointer for each list. The head pointer points to the first item in the list. The tail pointer points to the last item in the list. In a singly-linked list the first item has a next pointer which points to the next item in the list which is then the second item in the list. The second item has also a next pointer which points at the item which follows next after the second item which is then the third item. The third item similarly has a next pointer which points at the fourth item, and so forth to the last item in the list. The last item also has a next pointer, but since it is the last item, its next pointer is typically null (all zeros).

Another type of list which offers advantages relative to speed of traversing the list is the doubly-linked list. In such a list each item has not only a next pointer but also a back pointer. The back pointer is used to point just ahead in the list. For example, the back pointer of the third item in the list points to the second item while the next pointer of the third item points to the fourth

item. The last item's next pointer is still null while its back pointer points to the next-to-last item, assuming of course that the list has more than one item. For the first item in the list the back pointer is null in view of there being no item ahead of it, and the next pointer points to the second item.

During the operation of the data processing system it is desirable to add and delete items from lists. One technique that has been developed in a mainframe data processing system for such an operation for singly-linked lists is an instruction called "compare and swap". The instruction can be used to cause the proper adjustment to the head pointer of the list so that an item can be added or deleted at the head of the list. Before the compare and swap instruction is excuted, however, the address in the head pointer is read and stored. Then if, for example, a new item is to be added, a swap value which can be substituted into the head pointer is prepared. This swap value when so substituted will cause the header pointer to point at the new item. The preparation of the new item includes causing its next pointer to point at the next following item which is the item which is at the head of the list prior to the addition of the new item. After the new item is ready, the stored address is compared to the address in the head pointer to make sure they are still the same. During the preparation of the new item, the list may have been altered causing the next pointer of the preceding item to also have been altered. If the compared addresses are different, the process must start over. If the compared addresses are the same, an address pointing to the new item (the swap value) is inserted into the next pointer of the preceding item, thus effecting the addition of the new item. As previously stated, lists often have tail and head pointers for assisting in keeping track of the lists. Such a compare and swap operation is not effective for adding or deleting at the end of the list even when the

list has head and tail pointers. This is because not only the tail pointer must be changed but also the former last item must be changed to point at the new last item. This has not been available. Similarly, insertion and deletion was not available in the middle of a list because the item that is to point at the new item (in the case of a new item being added) may have been moved to a different list. Consequently, the compare test may be passed but the new item would be inserted into the wrong list.

Summary of the Invention

An object of the subject invention is to provide an improved compare and swap instruction.

Another object of the invention is to provide an improved technique for changing two values in a data processing system.

Yet another object of the invention is to provide an improved technique for manipulating linked lists in a data processing system.

These and other objects of the invention are achieved in a data processor in which two values are to be changed effectively simultaneously. A first compare value is read at a first location where a value is to be changed. A second compare value is read at a second location where a value is to be changed. A first swap value is stored which is the value to be substituted for the value at the first location. A second swap value is stored which is the value to be substituted for the value at the second location. An instruction which is uninterruptable can then commence. The instruction compares the first compare value to the value present at the first location, and the second compare value to the value present at the second location. If the first and second compare values are the same as the respective values at the first and second location, the

first swap value is input to the first location, and the second swap value is input to the second location.

Brief Description of the Drawings

FIG. 1 is a block diagram of a data processing system useful for implementing the invention;

FIG. 2 is a block diagram of the data processor of FIG. 1;

FIG.s 3A and 3B, placed top to bottom, comprise a micro-control flow diagram according to a preferred embodiment of the invention;

FIG.s 4A and 4B are diagrams of doubly-linked lists useful for understanding how to use the present invention to add or delete an item at an end of a doubly-linked list; and

FIG.s 5A and 5B are diagrams of singly-linked lists useful for understanding how to add or delete an item at the middle of a singly-linked list.

Description of the Invention

Shown in Figure 1 is a data processing system 10 wherein logical addresses (LADDR) issued by a data processor (DP) 12 are mapped by a memory management unit (MMU) 14 to a corresponding physical address (PADDR) for output on a physical bus (PBUS) 16. Simultaneously, the various logical access control signals (LCNTL) provided by DP 12 to control the access are converted to appropriately timed physical access control signals (PCNTL) by a modifier unit 18 under the control of MMU 14. DP 12 is an example of a data processor which is capable of implementing the present invention relating to compare and swap instructions.

In response to a particular range of physical addresses (PADDR), memory 20 will cooperate with an error detection and correction circuit (EDAC) 22 to exchange data (DATA) with DP 12 in synchronization with the physical access control signals (PCNTL) on PBUS 16. Upon detecting an error in the data, EDAC 22 will either signal a bus error (BERR) or request DP 12 to retry (RETRY) the exchange, depending upon the type of error.

In response to a different physical address, mass storage interface 24 will cooperate with MP 12 to transfer data to or from mass storage 26. If an error occurs during the transfer, interface 24 may signal a bus error (BERR) or, if appropriate, request a retry (RETRY).

In the event that the MMU 14 is unable to map a particular logic address (LADDR) into a corresponding physical address (PADDR), the MMU 14 will signal an access fault (FAULT). As a check for MMU 14, a watchdog timer 28 may be provided to signal a bus error (BERR) if no physical device has responded to a physical address (PADDR) within a suitable time period relative to the physical access control signals (PCNTL).

If, during a data access bus cycle, a RETRY is requested, OR gates 30 and 32 will respectively activate the BERR and HALT inputs of DP 12. In response to the simultaneous activation of both the BERR and HALT inputs thereof during a DP-controlled bus cycle, DP 12 will abort the current bus cycle and, upon the termination of the RETRY signal, retry the cycle.

If desired, operation of DP 12 may be externally controlled by judicious use of a HALT signal. In response to the activation of only the HALT input thereof via OR gate 32, DP 12 will halt at the end of the current bus cycle, and will resume operation only upon the termination of the HALT signal.

In response to the activation of only the BERR input thereof during a processor-controlled bus cycle, DP 12 will abort the current bus cycle, internally save the contents of the status register, enter the supervisor state, turn off the trace state if on, and generate a bus error vector number. DP 12 will then stack into a supervisor stack area in memory 20 a block of information which reflects the current internal context of the processor, and then use the vector number to branch to an error handling portion of the supervisor program.

Up to this point, the operation of DP 12 is identical to the operation of Motorola's MC68000 microprocessor. However, DP 12 differs from the MC68000 in the amount of information which is stacked in response to the assertion of BERR. The information stacked by the MC68000 consists of: the saved status register, the current contents of the program counter, the contents of the instruction register which is usually the first word of the currently executing instruction, the logical address which was being accessed by the aborted bus cycle, and the characteristics of the aborted bus cycle, i.e. read/write, instruction/data and function code. In addition to the above information, DP 12 is constructed to stack much more information about the internal machine state. If the exception handler is successful in resolving the error, the last instruction thereof will return control of DP 12 to the aborted program. During the execution of this instruction, the additional stacked information is retrieved and loaded into the appropriate portions of DP 12 to restore the state which existed at the time the bus error occurred.

The preferred operation of DP 12 will be described with reference to Figure 2 which illustrates the internal organization of a microprogrammable embodiment of DP 12. Since the illustrated form of DP 12 is very similar to the Motorola MC68000 microprocessor described in detail in the several U.S. Patents cited hereafter, the common operation

aspects will be described rather broadly. Once a general understanding of the internal architecture of DP 12 is established, the discussion will focus on the unique compare and swap feature of the present invention.

The DP 12, like the MC68000, is a pipelined, microprogrammed data processor. In a pipelined processor, each instruction is typically fetched during the execution of the preceding instruction, and the interpretation of the fetched instruction usually begins before the end of the preceding instruction. In a microprogrammed data processor, each instruction is typically fetched during the execution of the preceding instruction, and the interpretation of the fetched instruction usually begins before the end of the preceding instruction. In a microprogrammed data processor, each instruction is executed as a sequence of microinstructions which perform small pieces of the operation defined by the instruction. If desired, user instructions may be thought of as macroinstructions to avoid confusion with the microinstructions. In the MC68000 and DP 12, each microinstruction comprises a microword which controls microinstruction sequencing and function code generation, and a corresponding nanoword which controls the actual routing of information between functional units and the actuation of special function units within DP 12. With this in mind, a typical instruction execution cycle will be described.

At an appropriate time during the execution of each instruction, a prefetch microinstruction will be executed. The microword portion thereof will, upon being loaded from micro ROM 34 into micro ROM output latch 36, enable function code buffers 38 to output a function code (FC) portion of the logical address (LADDR) indicating an instruction cycle. Upon being simultaneously loaded from nano ROM 40 into nano ROM output latch 42, the corresponding nanoword requests bus controller 44 to

perform an instruction fetch bus cycle, and instructs execution unit 46 to provide the logical address of the first word of the next instruction to address buffers 48. Upon obtaining control of the PBUS 16, bus controller 44 will enable address buffers 48 to output the address portion of the logical address (LADDR). Shortly thereafter, bus controller 44 will provide appropriate data strobes (some of the LCNTL signals) to activate memory 20. When the memory 20 has provided the requested information, bus controller 44 enables instruction register capture (IRC) 50 to input the first word of the next instruction from PBUS 16. At a later point in the execution of the current instruction, another microinstruction will be executed to transfer the first word of the next instruction from IRC 50 into instruction register (IR) 52, and to load the next word from memory 20 into IRC 50. Depending upon the type of instruction in IR 52, the word in IRC 50 may be immediate data, the address of an operand, or the first word of a subsequent instruction. Details of the instruction set and the microinstruction sequences thereof are setforth fully in U.S. Patent No. 4,325,121 entitled "Two Level Control Store for Microprogrammed Data Processor" issued 13 April 1982 to Gunter et al, and which is hereby incorporated by reference.

As soon as the first word of the next instruction has been loaded into IR 52, address 1 decoder 54 begins decoding certain control fields in the instruction to determine the micro address of the first microinstruction in the initial microsequence of the particular instruction in IR 52. Simultaneously, illegal instruction decoder 56 will begin examining the format of the instruction in IR 52. If the format is determined to be incorrect, illegal instruction decoder 56 will provide the micro address of the first microinstruction of an illegal instruction microsequence. In response to the format error, exception logic 58 will force multiplexor 60 to substitute the micro

address provided by illegal instruction decoder 56 for the micro address provide by address 1 decoder 54. Thus, upon execution of the last microinstruction of the currently executing instruction, the microword portion thereof may enable multiplexor 60 to provide to an appropriate micro address to micro address latch 62, while the nanoword portion thereof enables instruction register decoder (IRD) 64 to load the first word of the next instruction from IR 52. Upon the selected micro address being loaded into micro address latch 62, micro ROM 34 will output a respective microword to micro ROM output latch 36 and nano ROM 40 will output a corresponding nanoword to nano ROM output latch 42.

Generally, a portion of each microword which is loaded into micro ROM output latch 36 specifies the micro address of the next microinstruction to be executed, while another portion determines which of the alternative micro addresses will be selected by multiplexor 60 for input to micro address latch 62. In certain instructions, more than one microsequence must be executed to accomplish the specified operation. These tasks, such as indirect address resolution, are generally specified using additional control fields within the instruction. The micro addresses of the first microinstructions for these additional microsequences are developed by address 2/3 decoder 66 using control information in IR 52. In the simpler form of such instructions, the first microsequence will typically perform some preparatory task and then enable multiplexor 60 to select the micro address of the microsequence which will perform the actual operation as developed by the address 3 portion of address 2/3 decoder 66. In more complex forms of such instructions, the first microsequence will perform the first preparatory task and then will enable multiplexor 60 to select the micro address of the next preparatory microsequence as developed by the address 2 portion of address 2/3 decoder 66. Upon performing this

additional preparatory task, the second microsequence then enables multiplexor 60 to select the micro address of the microsequence which will perform the actual operation as developed by the address 3 portion of address 2/3 decoder 66. In any event, the last microinstruction in the last microsequence of each instruction will enable multiplexor 60 to select the micro address of the first microinstruction of the next instruction as developed by address 1 decoder 54. In this manner, execution of each instruction will process through an appropriate sequence of microinstructions. A more thorough explanation of the micro address sequence selection mechanism is given in U.S. Patent No. 4,342, 078 entitled "Instruction Register Sequence Decoder for Microprogrammed Data Processor" issued 27 July 1982 to Tredennick et al, and which is hereby incorporated by reference.

In contrast to the microwords, the nanowords which are loaded into nano ROM output latch 42 indirectly control the routing of operands into and, if necessary, between the several registers in the execution unit 46 by exercising control over register control (high) 68 and register control (low and data) 70. In certain circumstances, the nanoword enables field translation unit 72 to extract particular bit fields from the instruction in IRD 64 for input to the execution unit 46. The nanowords also indirectly control effective address calculations and actual operand calculations within the execution unit 46 by exercising control over AU control 74 and ALU control 76. In appropriate circumstances, the nanowords enable ALU control 76 to store into status register (SR) 78 the condition codes which result from each operand calculation by execution unit 46. A more detailed explanation of ALU control 76 is given in U.S. Patent No. 4,312,034 entitled "ALU and Condition Code Control Unit for Data Processor" issued 19 January 1982 to Gunter, et al, and which is hereby incorporated by reference.

A read-modify-write cycle (RMC) is available which ensures in advance that a write can follow a read without relinquishing the bus. This RMC mode is indicated by an RMC signal. During the RMC mode no other system resource can have access to the bus. The RMC mode and functions involved therewith are described in U.S. Patent No. 4,348,722, entitled "Bus Processor", issued Sept. 7, 1982 to Gunter et al, and which is hereby incorporated by reference. The RMC signal can be brought out to an external pin to prevent system resources from attempting to gain access to the bus when the RMC signal is present. Additionally, in the preferred embodiment the RMC signal is controlled by microcode. Consequently, instructions can be made available which monopolize the bus for even longer than the two bus cycles required for a read followed by a write.

Such an instruction which can thus be made available is the compare and swap instruction for singly-linked lists (CAS1) as previously described. The CAS1 instruction, however, is not helpful for doubly-linked lists. A compare and swap instruction for doubly-linked lists (CAS2) is the subject of the present invention. Shown in FIG.s 3A and 3B, placed top to bottom, is a micro-control flow diagram for performing the CAS2 instruction according to a preferred embodiment of the invention.

Shown in FIG. 4A is a list 400 comprised of an item 401, an item 402, and an item 403. Shown in FIG. 4B is a list 400' which further includes an item 404 as well as items 401-403. Each item 401-404 has a back pointer and a next pointer. The back pointers of items 401-404 are 406, 407, 408, and 409, respectively. The next pointers of items 401-404 are 411, 412, 413, and 414, respectively. List 400 also includes a head pointer 416 and a tail pointer 417. In list 400 item 401 is the head. Accordingly, head pointer 416 points to item 401. As the head of the list 400, back pointer 406 of item 401 is null,

and next pointer of item 401 points to item 402. Back pointer 407 of item 402 points at item 401, and next pointer 412 points at item 403. With just three items in list 400, item 403 is the last item in list 400. Accordingly next pointer 413 of item 403 is null while its back pointer points at item 402. Tail pointer 417 points at item 403 because it is the last item in the list. Next pointer 411-413 and head pointer 416 form a first set of pointers which have an analogous function. Beginning with head pointer 416, list 400 can be traversed in a forward direction using this set of pointers. Tail pointer 417 and back pointers 406-408 form a second set of pointers. Beginning with tail pointer 417, list 400 can be traversed in a backward direction using the second set of pointers.

List 400' is the same as list 400 except that item 404 is the last item in the list. Consequently, tail pointer 417 points at item 404 instead of item 403 and next pointer 413 of item 403 points at item 404 instead of being null. Back pointer 409 of item 404 points at item 403, and next pointer 414 is null. Lists 400 and 400' are both properly formed doubly-linked lists with head and tail pointers. List 400' can be viewed as list 400 with a new item added at the tail. Alternatively list 400 can be viewed as list 400' but with the tail item deleted. A function of the CAS2 instruction is to be able to either add or delete an item at the head or tail using uninterruptable cycles. The first explanation is for adding an item at the tail. The list is originally list 400 which is changed by the CAS2 instruction to become list 400'.

Before the CAS2 instruction is performed, item 404 must be properly formed. As part of the formation process, item 404 has its back pointer 409 point to item 403 and next pointer 414 null. Also, prior to the CAS2 instruction, tail pointer 417 and next pointer 413 must be read and stored. Another step prior to beginning the CAS2 instruction is to store the values which are to be put into

tail pointer 417 and next pointer 413 so that these pointers will point to item 404. These values are sometimes referred to as swap values. As shown in FIG.s 1 and 2 there are numerous registers available for storing these swap values. These two pointers are the two that must be changed upon the addition of item 404 to list 400 in order to form list 400'. Tail pointer 417 and next pointer 413 must both be changed to point at item 404. During the formation of item 404 and after the reading of tail pointer 417 and next pointer 413, list 400 may be changed so that item 403 is no longer the last item in list 400. If that is the case, then at least one of tail pointer 417 and next pointer 413 will change. If such a change has occurred then item 404 cannot be properly added. The CAS2 instruction ensures that nothing has changed which will prevent the proper addition of item 404 before updating tail pointer 417 and next pointer 413. Another characteristic of the CAS2 instruction is that it will continue to completion even if an interrupt is received during the instruction. An instruction having this characteristic is often called an uninterruptable instruction.

To begin the CAS2 instruction, the RMC mode is instituted and the RMC signal is generated to ensure that the CAS2 instruction has exclusive use of the bus until the instruction is terminated. This prevents any changes to list 400 except those caused by the CAS2 instruction itself. The values at tail pointer 417 and next pointer 413 are then consecutively read. These two values are then compared to the corresponding values previously stored. This can be achieved by subtracting one from the other. If the result after subtraction is zero, then the values are the same. If the result is other than zero, then the values are different. If the values are the same, then the stored values for pointing to item 404 are swapped for those values in next pointer 413 and tail pointer 417.

After this swap the result is shown in FIG. 4B as list 400' with item 404 at the end. If either of the values are different, the RMC mode is terminated and the values last read from tail pointer 417 and next pointer 413 are stored in registers. In view of the changed circumstances as indicated by the values being different, a decision is then been made available as to the disposition of item 404. If item 404 is still to be added, a new CAS2 instruction can be instituted after pointers 409 and 414 have been properly updated to reflect the changed conditions.

Another possibility is for list 400' to be the beginning point and for the last item in the list, item 404, to be deleted . In such case list 400 shown in FIG. 4A is the desired result. As in the case for adding an item as previously described, some steps must be taken before the CAS2 instruction begins. In this deletion case, next pointer 413 is to swap its value which points to item 404 for the null value and tail pointer 417 is to swap its value which points to item 404 for a value which points to item 403. In preparation for this swap, the swap values comprising the null value and the value which points to item 403 are stored in registers. Also stored in registers are the compare values which comprise the value in tail pointer 417 which points to item 404 and the value in next pointer 413 which also points to item 404. The CAS2 instruction is then begun by initiating the RMC mode. The test values comprising the values in next pointer 413 and tail pointer 417 are then compared to the corresponding stored compare values. If each compare value is the same as its corresponding test value, the swap values are swapped for the corresponding values in next pointer 413 and tail pointer 417. The result of this swap is list 400 shown in FIG. 4A. If either of the compare values is different than its corresponding test value, then the swap does not occur, the RMC mode is terminated, and the new values in next pointer 413 and tail pointer 417 replace the previous compare values stored in registers.

The CAS2 instruction itself which is used to achieve a deletion or addition of an item is shown in FIG.s 3A and 3B. Compare values and swap values must be established prior to beginning the CAS2 instruction. The compare values are the ones which are read from the locations which need to be changed in order to properly effect the addition or deletion of the particular item. The swap values are those which are to replace the values in the locations which are to have their values changed. After initiating the RMC mode, a first test value is read from a first of the locations to be changed. A second test value is read from the other location while the first test value is compared to its corresponding compare value. A compare value is said to be corresponding when it was read from the same location as the test value to which it was said to be corresponding. The second test value is then compared to its corresponding compare value if the first test value is the same as its corresponding compare value. If not, the RMC mode is terminated. If the RMC mode is not terminated and if the comparison of the second test value to the second compare value shows that these two values are the same, then the swap value for the location of the first compare and test values is substituted for the first test value at that location. The other swap value then replaces the second test value at that location. If the second test value is not the same as the second test value, then the RMC is terminated at that point so that no swap values are swapped with the test values. Consequently, any two values can be safely swapped ina single instruction. The use is not limited to that described for FIG. 4A and FIG. 4B.

Shown in FIG. 5A is a list 500 comprised of an item 501, an item 502, an item 503, a head pointer 504, a tail pointer 505 and an access counter 506. Access counter 506 has a value which is incremented every time that a change is made to list 500. The value is read or incremented on a bidirectional bus 508. Each item 501-503 has a next

pointer 509, 510, and 511, respectively. Next pointer 509 of item 501 points at item 502, next pointer 510 of item 502 points at item 502, and next pointer 511 of item 503 is null. Head pointer 504 points at item 501 which is thus a head item. Tail pointer 505 points at item 503 which is thus a tail item. List 500 is a properly formed singly-linked list having head and tail pointers and an access counter. List 500' shown in FIG. 5B is the same as list 500 except that an item 512 has been inserted between items 502 and 503. With item 512 so inserted, next pointer 510 points at item 512 instead of item 503 as it did in list 500. Item 512 has a next pointer 513 pointing at item 503. List 500' is thus a properly formed singly-linked list with one additional item to that of list 500, the additional item being inserted in the middle of the list between the head and tail items 501 and 503. Whereas FIG.s 4A and 4B are useful for describing the addition and deletion of an item at an end of a list, FIG.s 5A and 5B are useful for describing the addition and deletion of an item in the middle of a list. For describing an addition or insertion of an item, FIG. 5A shows before and FIG. 5B shows after the insertion. Conversely, for a deletion, FIG. 5B shows before and FIG. 5A shows after.

The insertion of item 512 into list 500 between items 502 and 503 will be described first. The changes that need to be made to list 500 in order to effect the insertion are to next pointer 510 of item 502 and the value in access counter 506. Instead of pointing at item 503, next pointer 510 will point to item 512. Prior to beginning the CAS2 instruction, compare values and swap values must be stored. The compare values to be stored are those in next pointer 510 which points to item 503, and access counter 506. The swap values are those to be inserted into next pointer 510 and provided to access counter 506 to effect the insertion of item 512. After the insertion next pointer 510 is to have a value which points to item 512,

and access counter 506 will be incremented. In order to properly prepare item 512 it is necessary to know which item is to be after 512 in order to establish where next pointer 513 is to point. After item 512 has then been properly prepared, the CAS2 instruction can commence. The RMC mode is initiated and next pointer 510 and access counter 506 are read. The values at next pointer 510 and access counter 506 which are read during the CAS2 instruction are check values. Before the swap values replace these check values in next pointer 510, it is necessary to determine that items 502 and 503 have retained their relationship to each other and that access counter 506 has not been incremented. While item 512 was being formed, one of items 502 or 503 may have been deleted. Alternatively, another item or items may have been inserted between items 502 and 503. Another possibility is that items 502 and 503 may have retained their relationship to each other but have been moved to another list. If something has changed, it would be improper to insert the swap values into next pointer 510 and access counter 506. Consequently, the check values are compared to the compare values to make certain they are the same before inserting the swap values. In particular, the value read at next pointer 510 before the CAS2 instruction commenced is compared to the value which is present at next pointer 510 after the CAS2 instruction has commenced, and the value read at access counter 506 before the CAS2 instruction commenced is compared to the value which is present after the CAS2 instruction at access counter 506. If the comparisons show that the check values equal the compare values, then the swap values are inserted into next pointer 510 and access counter 506 so that next pointer 510 points at item 512 and that access counter 506 is incremented. When such insertion is made, the result is list 500' shown in FIG. 5B. If either of the comparisons results in showing that one of the check values is not the same as

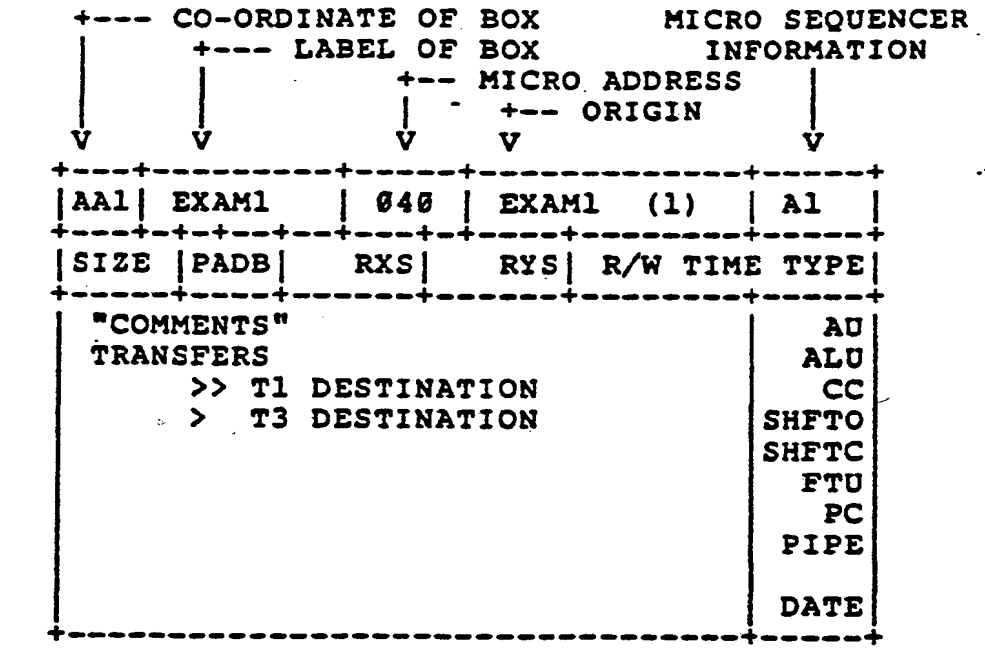
its corresponding compare value, neither swap value is inserted. The comparisons are consecutively done, then if appropriate in view of the comparisons, the insertions of the swap values are also consecutively done.

The CAS2 instruction is also useful for deleting an item from the middle of a list, such as deleting item 512 from list 500' to obtain list 500. The compare values which are read prior to commencing the CAS2 instruction are those at next pointer 510 and access counter 506. The swap value for next pointer 510 will be that for pointing at item 503. The swap value for access counter 506 will be the incremented value of the access counter 506. When the CAS2 instruction commences the RMC mode is initiated. A first test value is read from either next pointer 510 or access counter 506. This first test value is then compared to its corresponding compare value, while a second test value is read from the other of next pointer 510 or access counter 506. If the first test value is not the same as its corresponding compare value, then the CAS2 instruction and RMC mode are terminated. If the first test value is the same as its corresponding compare value, then the second test value is compared to its corresponding compare value. If the second test value is not the same as its corresponding compare value, the CAS2 instruction and the RMC mode are terminated, and the first and second test values replace the compare values. If the second test value is the same as its corresponding compare value, the swap values are consecutively swapped for the values in next pointer 510 and access counter 506. In particular, the swap value which is inserted into next pointer 510 causes next pointer 510 to point at item 503, and the swap value which is inserted into access counter 506 is the incremented value of the access counter 506. Whether next pointer 510 or access counter 506 is updated before the other is not significant. After effecting the insertion of the swap values, the CAS2 instruction and RMC mode are terminated. The CAS2 instruction thus effects the deletion of item 512 from list 500' to obtain list 500.

The CAS2 instruction is not limited to safely swapping 2 values in a linked list situation. The instruction can be useful anytime it is desirable to simultaneously swap 2 values. Such swapping can involve values of other types of list, but may also involve values unrelated to lists. The CAS2 instruction may be useful in any data structure which can have one location changed in conjunction with an acccess counter. A tree structure is such a situation where it can be useful to swap 2 such values effectively simultaneously.

The microcode for achieving this CAS2 instruction is disclosed in appendices I and II.

APPENDIX I
MICROINSTRUCTION LISTING



ORIGIN: if shared, co-ordinate of origin
if origin, # of boxes sharing with this box

DATA ACCESS INFORMATION:

R/W
- . - no access
- <W> - write
- <> - read
- SPC - special signal
- EXL - latch exception

TIME
- X - no timing associated
- T1 - write to aob in T1
- T3 - write to aob in T3
- T0 - aob writen before T1

TYPE

.,<>,<W> on R/W
- . - normal access
- UNK - program/data access
- CNORM - conditional normal
- CUNK - conditional prog/data
- AS - alternate address space
- CPU1 - cpu access - different bus error
- CPU2 - cpu access - normal bus error
- RMC - read-modify-write access

SPC on R/W
- RST1 - restore stage 1
- RST2 - restore stage 2
- HALT - halt pin active
- RSET - reset pin active
- SYNC - synchronize machine

EXL on R/W
- BERR - bus error
- AERR - address error
- PRIV - privilege viol.
- TRAC - trace

LINA	- line a 21	TRAP	- trap
LINF	- line f	COP	- protocol viol.
ILL	- illegal	FORE	- fomat error
DVBZ	- divide by zero	INT	- interrupt 1st stack
BDCK	- bad check	INT2	- interrupt 2nd stack
TRPV	- trap on overflow	NOEX	- no exception

MICRO SEQUENCER INFORMATION:-

DB - direct branch - next microaddress in microword
BC - conditional branch
A1 - use the A1 PLA sample interrupts and trace
A1A - use the A1 PLA sample interrupts, do not sample trace
A1B - use the A1 PLA do not sample interrupts or trace
A2 - use the A2 PLA
A7 - functional conditional branch (DB or A2 PLA)
A4 - use the A4 latch as next micro address
A5 - use the A5 PLA
A6 - use the A6 PLA

SIZE:

size = byte	nano specified constant value
size = word	nano specified constant value
size = long	nano specified constant value
size = ircsz	irc[11]=0/1 => word/long
size = irsz	ird decode of the instruction size (byte/word/long). Need to have file specifying residual control
size = ssize	shifter control generates a size value. The latch in which this value is held has the following encoding 000 = byte 001 = word 010 = 3-byte 011 = long 100 = 5-byte *** must act as long sized

RXS - RX SUBSTITUTIONS:

RX is a general register pointer. It is used to point at either special purpose registers or user registers. RX generally is used to translate a register pointer field within an instruction into the control required to select the the appropriate register.

rx = rz2d/rxd	conditionally substitute rz2d use rz2d and force rx[3]=0
	mul.l 0100 110 000 xxx xxx
	div.l 0100 110 001 xxx xxx

rx = rx	ird[11:9] muxed onto rx[2:0] rx[3] = 0 (data reg.) (unless residual points) rxa then rx[3] = 1 (residual defined in opmap)
rx = rz2	irc2[15:12] muxed onto rx[3:0] rx[3] is forced to 0 by residual control div.l 0100 110 001 xxx xxx bit field reg 1110 1xx 111 xxx xxx
rx = rp	rx[3:0] = ar[3:0] The value in the ar latch must be inverted before going onto the rx bus for movem rl,-(ry) 0100 100 01x 100 xxx
rx = rz	irc[15:12] muxed onto rx[3:0] (cannot use residual control)
rx = ro2	rx[2:0] = irc2[8:6] rx[3] = 0 (data reg.) Used in Bit Field, always data reg
rx = car	points @ cache address register
rx = vbr	points @ vector base register
rx = vatl	points @ vatl
rx = dt	points @ dt
rx = crp	rx[3:0] = ar[3:0] The value in ar points at a control register (i.e. not an element of the user visible register array)
rx = usp	rx[3:0] = F force effect of psws to be negated (0)
rx = sp	rx[2:0] = F, if psws=0 then address usp if psws=1 & pswm=0 then isp if psws=1 & pswm=1 then msp

RYS - RY SUBSTITUTIONS:

ry = ry	ird[2:0] muxed onto ry[2:0] ry[3] = 1 (addr reg.) unless residual points ryd then ry[3] = 0. (residual defined in opmap)
ry = ry/dbin ry/dob	This is a conditional substitution for the normal ry selection (which

includes the residual substitutions like dt) with dbin or dob. The substitution is made based on residual control defined in opmap (about 2 ird lines) which selects the dbin/dob and inhibits all action to ry (or the residually defined ry). Depending upon the direction to/from the rails dbin or dob is selected. If the transfer is to the rails then dbin is substituted while if the transfer is from the rails dob is substituted.

Special case: IRD = 0100 0xx 0ss 000 xxx (clr,neg,negx,not) where if driven onto the a-bus will also drive onto the d-bus.

ry = rw2 irc2[3:0] muxed onto ry[3:0]

use rw2

movem ea,rl	0100 110 01x xxx xxx
div.l	0100 110 001 xxx xxx
bfield	1110 xxx xxx xxx xxx
cop	1111 xxx xxx xxx xxx

do not allow register to be written to

div.w	1000 xxx x11 xxx xxx

force ry[3] = 0

div.l	0100 110 001 xxx xxx
bfield	1110 1xx x11 xxx xxx

ry = rw2/dt conditionally substitute rw2 or dt

use rw2 and force ry[3]=0

mul.l	0100 110 000 xxx xxx and irc2[10] = 1
div.l	0100 110 001 xxx xxx and irc2[10] = 1

ry = vdtl points @ virtual data temporary

ry = vat2 points @ virtual address temporary 2

ry = dty points @ dt

AU - ARITHMETIC UNIT OPERATIONS:

0- ASDEC add/sub add/sub based on residual control sub if ird = xxxx xxx xxx 100 xxx

1- ASXS add/sub add/sub based on residual (use alu add/sub). Do not extend db entry

add if ird = 1101 xxx xxx xxx xxx add

or - 0101 xxx 0xx xxx xxx addq

2- SUB	sub	subtract AB from DB
3- DIV	add/sub	do add if aut[31] = 1, sub if aut[31] = 0; take db (part rem) shift by 1 shifting in alut[31] then do the add/sub.
4- NIL		
6- SUBZX	sub	zero extend DB according to size then sub AB
8- ADDX8	add	sign extend DB 8 -> 32 bits then add to AB
9- ADDX6	add	sign extend DB 16 -> 32 bits then add to AB
10- ADD	add	add AB to DB
11- MULT	add	shift DB by 2 then add constant sign/zero extend based on residual and previous aluop muls = always sxtd mulu = sxtd when sub in previous aluop
12- ADDXS	add	sign extend DB based on size then add to AB
13- ADDSE	add	sign extend DB based on size then shift the extended result by 0,1,2,3 bits depending upon irc[10:9]. Finally add this to AB
14- ADDZX	add	zero extend DB according to size then add to AB
15- ADDSZ	add	zero extend DB according to size, shift by 2, then add

CONSTANTS

0,1	1 selected by: (div * allzero) + (mult * alu carry = 0)
1,2,3,4	selected by size byte = 1 word = 2 3-by = 3 long = 4

If (Rx=SP or Ry=SP) and (Ry=Ry or Rx=Rx) and (Rx or Ry is a source and destination) and (au constant = 1,2,3,4) and (size = byte) then constant = 2 rather than one.

ALU - ARITHMETIC AND LOGIC UNIT OPERATIONS:

col0 = x,nil
col1 = and
col2 = alu1,div,mult,or
col3 = alu2,sub

row		col 1	col 2	col 3
1	ADDROW	and	add	
2	ADDXROW	and	addx	add
3	SUBROW	and	sub	
4	SUBXROW	and	subx	add1
5	DIVROW	and	div	sub
6	MULTROW	and	mult	sub
7	ANDROW	and	and	
8	EORROW	and	eor	
9	ORROW	and	or	add
10	NOTROW	and	not	
11	CHGROW	and	chg	
12	CLRROW	and	clr	
13	SETROW	and	set	

		cin
add	db + ab	0
addx	db + ab	x
add1	db + ab	1
and	ab ^ db	-
chg	ab xor k=-1	-
clr	ab ^ k=0	-
eor	ab xor db	-
not	~ab v db	-
or	ab v db	-
set	ab v k=-1	-
sub	db + ~ab	1
subx	db + ~ab	x

mult (db shifted by 2) add/sub (ab shifted by 0,1,2 (if 0 then add/sub 0)) control for add/sub and shift amount comes from regb. Don't assert atrue for mult

cin = 0

div build part. quot and advance part. remain.1 ab (pr.1:pq) shifted by 1, add0, value shifted in = au carry (quot bit)

cin = 0

must assert atrue for div

The condition codes are updated during late T3 based upon the data in alut and/or rega. These registers can be written to during T3. In the case of rega, there are times when the value to be tested is the result of an insertion from regb.

CC - CONDITION CODE UPDATE CONTROL:

row		col 1		col 2		col 3	
1	add	cnzvc		ddddd		ddddd	
2	addx	cnzvc		ddkdc	(bcd1)	cdzdc	(bcd2)
3	sub	cnzvc		knzvc	(cmp)	ddddd	
4	subx	cnzvc		ddkdc	(bcd1)	cdzdc	(bcd2)
5	div	knzv0	(div)	ddddd		ddddd	
6	mull	knzv0		ddddd		ddddd	
7	rotat	knz0c		ddddd		ddddd	
8	rox	cnz0c		knz00		kkkvk	
9	bit,bitfld	kkzkk	(bit)	knz00	(bfld1)	kkzkk	(bfld2)
10	log	knz00		ddddd		ddddd	

```
standard
  n = alut msb (by size)
  z = alut=0 (by size)

non-standard
  add      c = cout
           v = vout
  addx.1   c = cout
           z = pswz ^ locz
           v = vout
  bcd1     c = cout
  bcd2     c = cout v pswc
           z = pswz ^ locz
  bfld1    n = shiftend
           z = all zero
  bfld2    z = pswz ^ allzero
  bit      z = allzero
  div      v = au carry out
  mull     n = (shiftend ^ irc2[10]) v
               (alut[31] ^ ~irc2[10])
           z = (alut=0 ^ shift allzero ^ irc2[10]) v
               (alut=0 ^ ~irc2[10])
           v = ~irc2[10] ^ ((irc2[11] ^ (~allzero ^
               ~alut[31]) v (~allone ^ alut[31])) v
               (~irc2[11] ^ ~allzero))
  rotat    c = shiftend = (sc=0 - 0  sc<>0 - end)
  rox.1    c = shiftend = (sc=0 - pswx  sc<>0 - end)
               ! can do this in two steps as knz0c where
               ! c=pswx and cnz0c where c=shiftend (not
               ! with share row with shift)
  rox.3    v = shift overflow = ((~allzero ^ sc>sz) v
               (~(allzero v allones) ^ sc<=sz))
```

```
                ! can simplify this if we don't share
                ! rows but it will cost another box
sub.1     c = ~cout
          v =  vout
sub.2     c = ~cout
          v =  vout
subx.1    c = ~cout
          z = pswz ^ locz
          v =  vout
subx.2    c = ~cout
subx.3    c = ~cout v pswc
          z = pswz ^ locz
```

The meaning and source of signals which are used to set the condition codes is listed below:

allzero = every bit in rega field = 0 where the field is defined as starting at the bit pointed to by start and ending (including) at the bit pointed to by end. (see shift control)

allone = every bit in rega field = 1 where the field is defined as starting at the bit pointed to by start and ending (including) at the bit pointed to by end. (see shift control)

shiftend = the bit in rega pointed to by end = 1. (see shift control)

locz = all alut for the applicable size = 0.

SHFTO - SHIFTER OPERATIONS:

ror — value in rega is rotated right by value in shift count register into regb.

sxtd — value in rega defined by start and end registers is sign extended to fill the undefined bits and that value is rotated right by the value in the shift count register. The result is in regb.

xxtd — value in rega defined by start and end registers is PSWX extended to fill the undefined bits and that value is rotated right by the value in the shift count register. The result is in regb.

zxtd — value in rega defined by start and end registers is zero extended to fill the undefined bits and that value is rotated right by the value in the shift count register. The result is in regb.

ins the value in regb is rotated left by the value in shift count register and then inserted into the field defined by the start and end register in rega. Bits in rega that are not defined by start and end are not modified.

boffs provides the byte offset in regb. If irc2[11]=1 then the offset is contained in R0 and as such rega should be sign extended from rega to regb using the values established in start, end, and shift count of 3,31,3 respectively. If irc2[11]=0 then the offset is contained in the immediate field and should be loaded from irc2[10:6] or probably more conveniently osr[4:0]. This value however should be shifted by 3 bits such that osr[4:3] are loaded onto regb[1:0] with zero zero extension of the remaining bits.

offs provides the offset in regb. If irc2[11]=1 then the offset is contained in R0 and as such DB>REGB should be allowed to take place. If irc2[11]=0 then the offset is contained in the immediate field and osr[4:0] should be loaded onto regb[4:0] with zero extension of the remaining bits.

SHFTC - SHIFTER CONTROL:

```
        {sbm1}
BIT     st = 0
bit     en = -1          (31)
mvp     sc = wr          (16,32)
swap    wr = BC[12:7]    (16,32)
callm   osr = x
        cnt = x

        {sbm2}
        st = wr - 8
        en = wr - 1
        sc = wr - 8
        wr = wr - 8
        osr = x
        cnt = x

        {sbm3}
        st = DB [5:0] mod sz
        en = DB [5:0] mod sz
        sc = 0
        wr = DB [5:0]
        osr = x
        cnt = x

        {sbm4}
        st = 0
        en = -1          (31)
        sc = wr
        wr = wr
        osr = x
        cnt = x

        {sbm5}
        st = x
        en = x
        sc = x
        wr = DB [7:2]
        osr = x
        cnt[1:0] = DB [1:0]

        {sbm6}
        st = 16
        en = 31
        sc = 16
        wr = wr - 1
        osr = x
        cnt = x

        {}
        st = x
        en = x
        sc = x
```

```
        wr = x
        osr = x
        cnt = x

        {mul1}                                  {mul2}
MUL     st = wr                                 st = wr - 2
mulw    en = -1 mod sz   (15,31)                en = wr
mull    sc = wr                                 sc = wr - 2
        wr = BC[12:7]    (14,30)                wr = wr - 2
        osr = x                                 osr = x
        cnt = x                                 cnt = x

        {mul3}                                  {mul4}
        st = 0                                  st = 0
        en = -1          (31)                   en = en
        sc = x                                  sc = x
        wr = x                                  wr = x
        osr = x                                 osr = x
        cnt = x                                 cnt = x

        {}                                      {mul6}
        st = x                                  st = 16
        en = x                                  en = 31
        sc = x                                  sc = 16
        wr = x                                  wr = x
        osr = x                                 osr = x
        cnt = x                                 cnt = x

        {}
        st = x
        en = x
        sc = x
        wr = x
        osr = x
        cnt = x

        {divw1}                                 {divw2}
divw    st = 0                                  st = 0
        en = 31                                 en = -1 mod sz   (15)
        sc = wr          (16)                   sc = 16
        wr = BC[12:7]    (16)                   wr = wr - 1
        osr = x                                 osr = x
        cnt = x                                 cnt = x

        {divw3}                                 {divw4}
        st = wr          (16)                   st = 0
        en = -1          (31)                   en = 31
        sc = wr          (16)                   sc = wr
        wr = BC[12:7]    (16)                   wr = x
        osr = x                                 osr = x
        cnt = x                                 cnt = x

        {divw5}                                 {divw6}
        st = 4                                  st = 16
```

```
        en = -1 mod size (7)          en = 31
        sc = 28                       sc = 16
        wr = x                        wr = x
        osr = x                       osr = x
        cnt = x                       cnt = x

        {divw7}
        st = st
        en = -1   (31)
        sc = 0
        wr = x
        osr = x
        cnt = x

        {div11}                       {div12}
divl    st = wr - 1       (31)        st = 0
        en = -1           (31)        en = -1          (31)
        sc = x                        sc = 0
        wr = BC[12:7]     (32)        wr = wr - 1
        osr = x                       osr = x
        cnt = x                       cnt = x

        {div13}                       {div14}
        st = 0                        st = 0
        en = -1           (31)        en = 31
        sc = 0                        sc = 0
        wr = x                        wr = x
        osr = x                       osr = x
        cnt = x                       cnt = x

        {}                            {div16}
        st = x                        st = 16
        en = x                        en = 31
        sc = x                        sc = 16
        wr = x                        wr = x
        osr = x                       osr = x
        cnt = x                       cnt = x

        {}
        st = x
        en = x
        sc = x
        wr = x
        osr = x
        cnt = x

        {}                            {}
unk     st = x                        st = x
        en = x                        en = x
        sc = x                        sc = x
        wr = x                        wr = x
        osr = x                       osr = x
        cnt = x                       cnt = x
```

```
{}
st = x
en = x
sc = x
wr = x
osr = x
cnt = x

{}
st = x
en = x
sc = x
wr = x
osr = x
cnt = x

{}
st = x
en = x
sc = x
wr = x
osr = x
cnt = x

{asl1}
st = ø
en = osr + ~wr
sc = ~wr + 1
wr = DB [5:ø] or BC[12:7]  (Q)
osr = BC[5:ø]    (8,16,32)
cnt = x

{asl3}
st = ø
en = osr - 1
sc = x
wr = wr
osr = x
cnt = x

{}
st = x
en = x
sc = x
wr = x
osr = x
cnt = x

{}
st = x
en = x
sc = x
wr = x
osr = x
```

asl

```
{}
st = x
en = x
sc = x
wr = x
osr = x
cnt = x

{unk6}
st = 16
en = 31
sc = 16
wr = x
osr = x
cnt = x

{asl2}
st = x
en = ~(wr-1) mod sz
sc = x
wr = wr
osr = osr
cnt = x

{asl4}
st = osr + ~wr
en = -1 mod sz
sc = x
wr = wr
osr = x
cnt = x

{asl6}
st = 16
en = 31
sc = 16
wr = x
osr = x
cnt = x
```

```
        cnt = x

        {asr1}                                  {asr2}
asr     st = wr                                 st = wr - 1
        en = osr - 1                            en = (wr - 1) mod sz
        sc = wr                                 sc = x
        wr = DB [5:0] or BC[12:7] (Q)           wr = wr
        osr = BC[5:0]   (8,16,32)               osr = osr
        cnt = x                                 cnt = x

        {asr3}                                  {}
        st = osr - 1                            st = x
        en = osr - 1                            en = x
        sc = x                                  sc = x
        wr = wr                                 wr = x
        osr = osr                               osr = x
        cnt = x                                 cnt = x

        {}                                      {asr6}
        st = x                                  st = 16
        en = x                                  en = 31
        sc = x                                  sc = 16
        wr = x                                  wr = x
        osr = x                                 osr = x
        cnt = x                                 cnt = x

        {}
        st = x
        en = x
        sc = x
        wr = x
        osr = x
        cnt = x

        {rotl1}                                 {rotl2}
rotl    st = osr                                st = x
        en = -1         (31)                    en = ~(wr - 1) mod sz
        sc = osr                                sc = x
        wr = DB [5:0] or BC[12:7] (Q)           wr = wr
        osr = BC[5:0]   (8,16,32)               osr = osr
        cnt = x                                 cnt = x

        {rotl3}                                 {}
        st = 0                                  st = x
        en = 31                                 en = x
        sc = ~(wr - 1) mod sz                   sc = x
        wr = wr                                 wr = x
        osr = osr                               osr = x
        cnt = x                                 cnt = x

        {}                                      {rotl6}
        st = x                                  st = 16
        en = x                                  en = 31
        sc = x                                  sc = 16
```

```
        wr = x                                  wr = x
        osr = x                                 osr = x
        cnt = x                                 cnt = x

        {}
        st = x
        en = x
        sc = x
        wr = x
        osr = x
        cnt = x

        {rotr1}                                 {rotr2}
rotr    st = osr                                st = x
        en = -1          (31)                   en = (wr - 1) mod sz
        sc = osr                                sc = x
        wr = DB [5:0] or BC[12:7] (Q)           wr = wr
        osr = BC[5:0]    (8,16,32)              osr = osr
        cnt = x                                 cnt = x

        {rotr3}                                 {}
        st = 0                                  st = x
        en = 31                                 en = x
        sc = wr mod sz                          sc = x
        wr = wr                                 wr = x
        osr = osr                               osr = x
        cnt = x                                 cnt = x

        {}                                      {rotr6}
        st = x                                  st = 16
        en = x                                  en = 31
        sc = x                                  sc = 16
        wr = x                                  wr = x
        osr = x                                 osr = x
        cnt = x                                 cnt = x

        {}
        st = x
        en = x
        sc = x
        wr = x
        osr = x
        cnt = x

        {rox11}                                 {rox12}
rox1    st = 0                                  st = 0
        en = osr + ~wr   (14)                   en = (osr - wr) mod sz
        sc = -1          (31)                   sc = 0
        wr = BC[12:7]    (1)                    wr = wr
        osr = BC[5:0]    (16)                   osr = osr
        cnt = x                                 cnt = x

        {rox13}                                 {rox14}
        st = (~(wr-1) + 1) mod sz               st = 0
```

```
        en = -1 mod sz
        sc = (~(wr-1) + 1) mod sz
        wr = DB [5:0] or BC[12:7]  (Q)
        osr = BC[5:0]    (8,16,32)
        cnt = x

        {rox15}
        st = (~(wr-1) + 1) mod sz
        en = -1 mod sz
        sc = (~(wr-1) + 1) mod sz
        wr = wr
        osr = osr
        cnt = x

        {rox16}
        st = 16
        en = 31
        sc = 16
        wr = wr - 1 - osr
        osr = osr
        cnt = x

        {rox17}
        st = wr - 1
        en = osr - 1
        sc = 0
        wr = wr
        osr = osr
        cnt = x

roxr    {roxr1}
        st = wr
        en = osr - 1
        sc = wr
        wr = BC[12:7]    (1)
        osr = BC[5:0]    (16)
        cnt = x

        {roxr2}
        st = 0
        en = (wr - 1) mod sz
        sc = 0
        wr = wr
        osr = osr
        cnt = x

        {roxr3}
        st = 0
        en = (wr-1) - 1
        sc = (wr-1) + 24,16,0
        wr = DB [5:0] or BC[12:7]  (Q)
        osr = BC[5:0]    (8,16,32)
        cnt = x

        {roxr4}
        st = wr
        en = osr - 1
        sc = wr
        wr = wr
        osr = osr
        cnt = x

        {roxr5}
        st = 0
        en = (wr-1) - 1
        sc = (wr-1) + 24,16,0
        wr = wr
        osr = osr
        cnt = x

        {roxr6}
        st = 16
        en = 31
        sc = 16
        wr = wr - 1 - osr
        osr = osr
        cnt = x

        {roxr7}
        st = 0
        en = osr - wr
        sc = 0
        wr = wr
        osr = osr
        cnt = x
```

bfreg

```
{bfrg1}
st = 0
en = 31
sc = osr + wr
wr = DB[4:0] or IRC2[4:0]
osr = REGB[4:0] or IRC2[10:6]
cnt = x

{bfrg2}
st = 0
en = wr - 1
sc = 0
wr = wr
osr = osr
cnt = x

{bfrg3}
st = 0
en = 31
sc = osr + wr
wr = wr
osr = osr
cnt = x

{}
st = x
en = x
sc = x
wr = x
osr = x
cnt = x

{bfrg5}
st = x
en = x
sc = x
wr = wr
osr = x
cnt[1:0] = DB [1:0]

{bfrg6}
st = 16
en = 31
sc = 16
wr = wr
osr = osr
cnt = x

{bfrg7}
st = 0
en = 31
sc = 25
wr = x
osr = x
cnt = x
```

bfmt

```
{bfmt1}
st = 3

en = -1          (31)

sc = 3
wr = DB[4:0] or IRC2[4:0]
osr = REGB[4:0] or IRC2[10:6]
cnt = x

{bfmt2}
st = 00:
     ~(osr[2:0]+(wr-1))
en = (osr[2:0]+(wr-1))
     [4:3]:~osr[2:0]
sc = 0
wr = wr
osr = osr
cnt = (osr[2:0]+
      (wr-1)) [5:3]

{bfmt3}
st = 0

en = 11:~osr[2:0]
sc = 0
wr = wr
osr = osr
cnt = x

{bfmt4}
st = 00:
     ~(osr[2:0]+(wr-1))
en = -1 mod sz   (7)
sc = 0
wr = wr
osr = x
cnt = x

{bfmt5}
st = x

{bfmt6}
st = 16
```

```
        en = x
        sc = x
        wr = x
        osr = x
        cnt = x

        {bfmt7}
        st = x
        en = x
        sc = x
        wr = x
        osr = x
        cnt = x

        {bfmi1}                                  {bfmi2}
bfmi    st = 3                                   st = 00:
                                                      ~(osr[2:0]+(wr-1))
        en = -1          (31)                    en = (osr[2:0]+(wr-1))
                                                      [4:3]:~osr[2:0]
        sc = 3                                   sc = 00:
                                                      ~(osr[2:0]+(wr-1))
        wr = DB[4:0] or IRC2[4:0]                wr = wr
        osr = REGB[4:0] or IRC2[10:6]            osr = osr
        cnt = x                                  cnt = (osr[2:0]+
                                                      (wr-1)) [5:3]

        {bfmi3}                                  {bfmi4}
        st = 0                                   st = 00:
                                                      ~(osr[2:0]+(wr-1))
        en = 11:~osr[2:0]                        en = -1 mod sz  (7)
        sc = 11:~(osr[2:0]+(wr-1))               sc = 00:
                                                      ~(osr[2:0]+(wr-1))
        wr = wr                                  wr = wr
        osr = osr                                osr = x
        cnt = x                                  cnt = x

        {bfmi5}                                  {bfmi6}
        st = 0                                   st = 16
        en = 00:(osr[2:0]+(wr-1))                en = 31
        sc = 25+(00:                             sc = 16
             (osr[2:0]+(wr-1)))
        wr = wr                                  wr = wr
        osr = x                                  osr = osr
        cnt[1:0] = DB [1:0]                      cnt = x

        {bfmi7}
        st = 0
        en = 31
        sc = 25
        wr = x
        osr = x
        cnt = x
```

```
cop     {cop1}
        st = x
        en = x
        sc = x
        wr = x
        osr = x
        cnt = x

        {cop2}
        st = x
        en = x
        sc = x
        wr = wr - 1
        osr = x
        cnt = x

        {cop3}
        st = x
        en = x
        sc = x
        wr = x
        osr = x
        cnt = x

        {cop4}
        st = x
        en = x
        sc = x
        wr = x
        osr = x
        cnt = x

        {cop5}
        st = x
        en = x
        sc = x
        wr = DB [7:2]
        osr = x
        cnt[1:0] = DB [1:0]

        {cop6}
        st = 16
        en = 31
        sc = 16
        wr = x
        osr = x
        cnt = x

        {cop7}
        st = x
        en = x
        sc = x
        wr = x
        osr = x
        cnt = x
```

1 loaded based on ird[5] - if ird[5] = 0 then wr value comes from BC bus else value is loaded from regc.

FTU - FIELD TRANSLATION UNIT OPERATIONS:

3- LDCR load the control register from regb. The register is selected by the value in ar[1:0], this can be gated onto the rx bus.

4- DPSW load the psw with the value in regb. Either the ccr or the psw is loaded depending upon size. If size = byte then only load the ccr portion.

14- CLRFP clear the f-trace pending latch. (fpend2 only)

17- LDSH2 load the contents of the shifter control registers from regb. These include wr,osr,count.

19- LDSWB load the internal bus register from regb. This is composed of bus controller state information which must be accessed by the user in fault situations.

21- LDSWI load the first word of sswi (internal status word) from regb. This is composed of tpend, fpend1, fpend2, ar latch

23- LDSH1 load the contents of the shifter control registers from regb. These include st,en,sc.

25- LDUPC load micro pc into A4 from regb and check validity of rev #.

26- LDPER load per with the value on the a-bus. (should be a T3 load). ab>per

28- LDARL load the ar latch from regb. May be able to share with ldswi or ldswj

29- ØPSWM clear the psw master bit.

33- RPER load output of per into ar latch and onto bc bus. There are two operations which use this function, MOVEM and BFFFO. MOVEM requires the least significant bit of the lower word (16-bits only) that is a one to be encoded and latched into the AR latch and onto the BC BUS (inverted) so that it can be used to point at a register. If no bits are one then the end signal should be active which is routed to the branch pla. After doing the encoding, the least significant bit should be cleared.

For BFFFO it is necessary to find the most significant bit of a long word that is a one. This value is encoded into 6 bits where the most significant bit is the 32-bit all zero signal. Thus the following bits would yield the corresponding encoding.

most sig bit set	per out	onto bc bus
31	Ø 11111	1110 0000
16	Ø 10000	1110 1111
Ø	Ø 00000	1111 1111
NONE	1 11111	0000 0000

The output is then gated onto the BC bus where it is sign extended to an 8-bit

value. It does not hurt anything in the BFFFO case to load the other latch (i.e. BFFFO can load the AR latch).
For BFFFO it does not matter if a bit is cleared.

34- STCR store the control register in regb. The register is selected by the value in ar[1:0], this can be gated onto the rx bus.

37- STPSW store the psw or the ccr in regb based on size. If size = byte then store ccr only with bits 8 - 15 as zeros.

38- 0PEND store the psw in regb then set the supervisor bit and clear the trace bit in the psw. Tpend and Fpend are cleared. The whole psw is stored in regb.

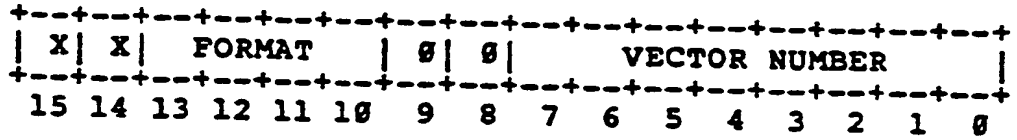
39- 1PSWS store the psw in regb then set the supervisor bit and clear both trace bits in the psw. The whole psw is stored in regb.

40- STINST store IRD decoded information onto the BC bus and into regb. This data can be latched from the BC bus into other latches (i.e. wr & osr) by other control.

41- STIRD store the ird in regb.

43- STINL store the new interrupt level in pswi and regb. The three bits are loaded into the corresponding pswi bits. The same three bits are loaded onto bc bus [3:1] with bc bus [31:4] = 1 and [0] = 1, which is loaded into regb. Clear IPEND the following T1.

44- STV# store the format & vector number associated with the exception in regb.



47- STCRC store the contents of the CRC register in regb. Latch A4 with microaddress.

48 STSH2 store the contents of the shifter control registers into regb. These include wr,osr,count. Store high portion of shift control

50- STSWB store the internal bus register in regb.

This is composed of bus controller state information which must be accessed by the user in fault situations.

52- STSWI store sswi (internal status word) in regb. The sswi is composed of tpend, ar latch, fpend1, fpend2

54- STSH1 store the contents of the shifter control registers into regb. These include st,en,sc.

56- STUPC store the micro pc in regb.

REV NUMBER	CRC	MICRO PC
15 14 13 12	11 10	9 8 7 6 5 4 3 2 1 0

62- NONE

63- STPER store the per onto the a-bus. (should be a T1 transfer). per>ab

PC - PC SECTION OPERATIONS:

	AOBP[1] = 0	AOBP[1] = 1
31 - 3PFI	EV3FI	OD3FI
30 - 3PFF	TPF	EV3FI

0- NF

```
aobpt>db>sas
tp2>ab>sas
```

1- TPF

```
aobpt>db>tp1
aobpt>db>aup>aobp*,aobpt
+2>aup
tp1>tp2
tp2>ab>sas
```

2- PCR

```
tp2>ab>a-sect
(if ry=pc then connect pc and address section)
aobpt>db>sas
```

3- PCRF

```
aobpt>db>tp1
aobpt>db>aup>aobp*,aobpt
+2>aup
tp1>tp2
tp2>ab>a-sect
```

```
        (if ry=pc then connect pc and address section)

   4- JMP1
          tp2>db>a-sect
          a-sect>ab>aobpt

   5- BOB
          aobpt>db>tp1
          tp1>tp2
          tp2>ab>sas

    - EV3FI
          aobpt>db>tp1*
          aobpt>db>aup>aobpt
          +4>aup
          tp2>ab>sas

    - OD3FI
          aobpt>db>aup>aobpt,tp2
          +2>aup
          tp2>ab>sas

   7- TRAP
          tp2>db>a-sect
          pc>ab>sas

   8- TRAP2
          tp2>ab>a-sect
          aobpt>db>sas

   9- JMP2
          a-sect>ab>aobpt
          aobpt>db>sas

  10- PCOUT
          pc>ab>a-sect
          aobpt>db>sas

  11- NPC   Conditional update based on cc=t/f
          tp2>db>aup,a-sect
          a-sect>ab>aup>aobpt

  12- LDTP2
          a-sect>ab>tp2
          aobpt>db>sas

  13- SAVE1
          pad>aobp
          aobpt>db>sas
          tp2>ab>sas

  15- SAVE2
          aobp>db>tp1
          tp2>ab>sas
```

```
14- FIX
        aobpt>db>tp1
        tp2>ab>aobpt
        tp1>tp2

16- LDPC
        tp2>pc
        aobpt>db>sas
        tp2>ab>sas

PIPE - PIPE OPERATIONS:

 Description of bit encodings.
     [6] = use irc
     [5] = change of flow
     [4] = fetch instruction
     [3:0] = previously defined pipe control
             functionality.
```

	AOBP[1] = 0	AOBP[1] = 1
0 1 1 3 -	3UDI	EV3Fa	OD3F
1 0 1 7 -	3UDF	TUD	EV3Fb

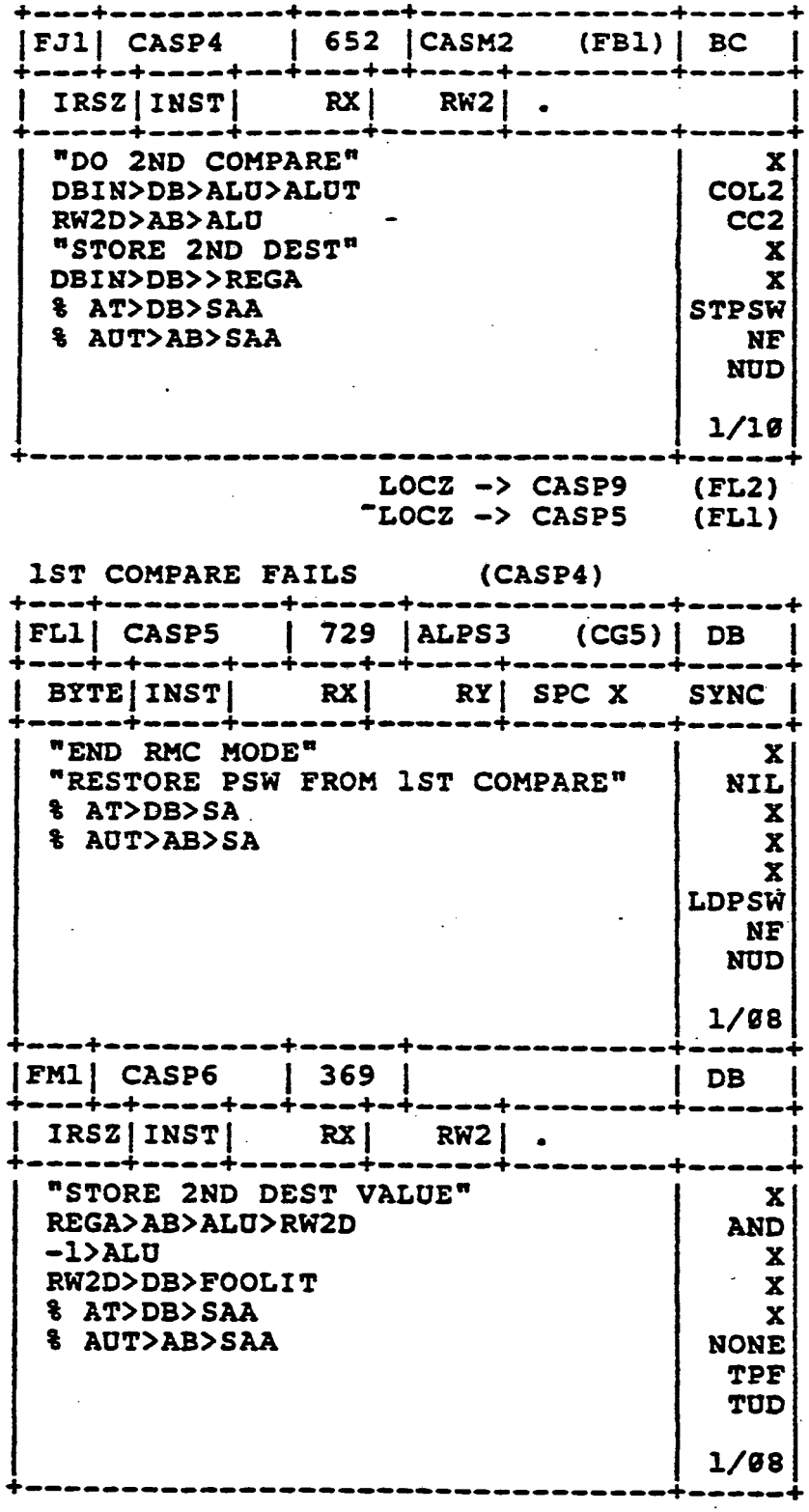
```
 - EV3Fa
        chr1>irb
        chrh>pb>imh,iml,irc
        change of flow
        fetch instr

 - EV3Fb
        chr1>irb
        chrh>pb>imh,iml,irc
        irc>ir                      ! implies use irc
        use pipe
        fetch instr

 - OD3F
        chr1>pb>irc
        ! force miss regardless of whether odd or even
        change of flow
        fetch instr

0 0 0 0 - NUD
        x

1 0 0 0 - UPIPE
        use pipe

0 0 1 1- FIX2                   Always transfer irb up pipe
```

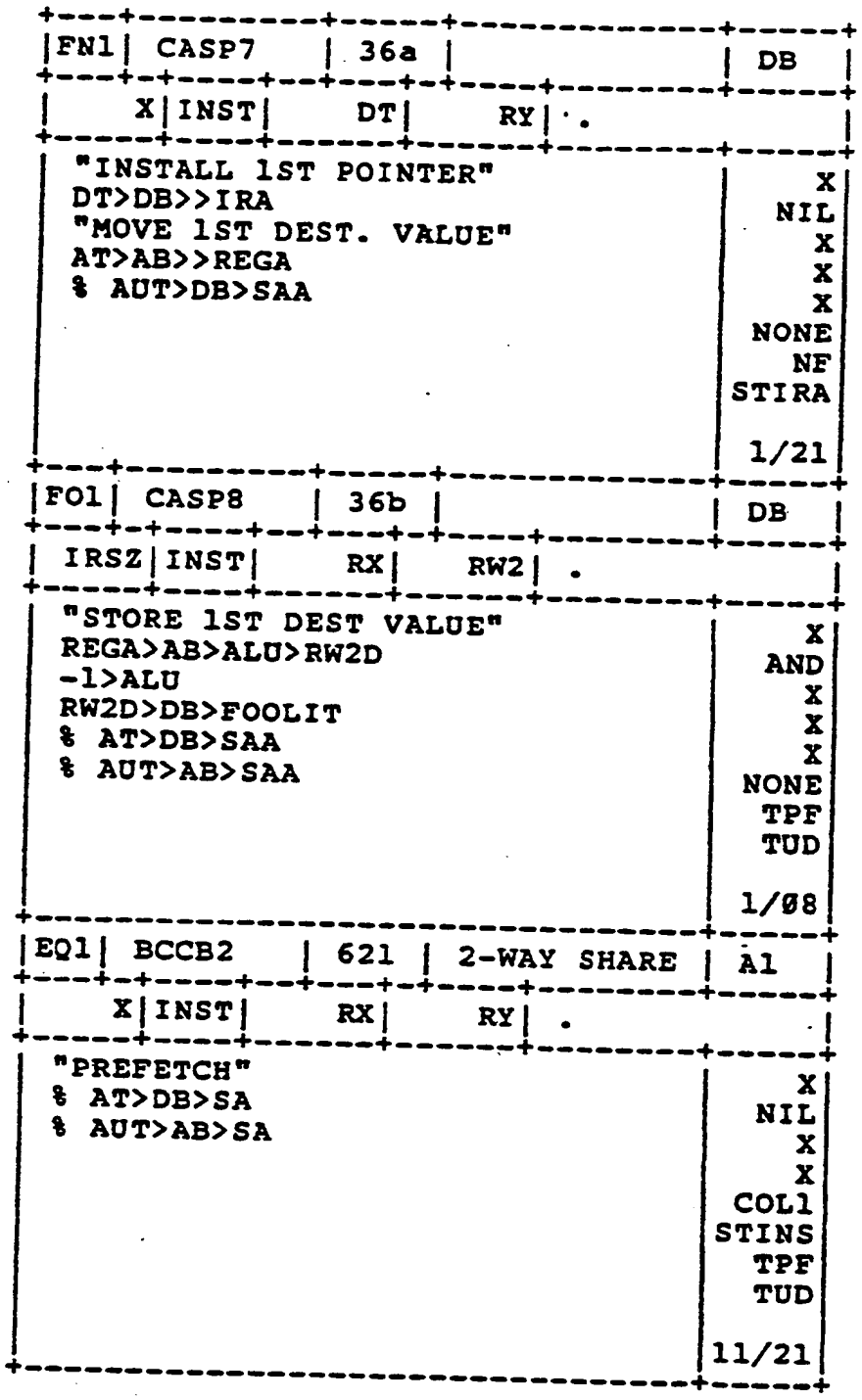
```
        chr>irb                 to irc,im and if irb needs
        irb>pb>imh,iml,irc      to be replaced, do access
                                and transfer chr to irb.
        ! force miss regardless of whether odd or even
        change of flow,
        fetch instr

        db>ird                  else load irb from d-bus.
        irb>pb>imh,iml,irc
        change of flow
        fetch instr

0 0 0 2 - IRAD
        ira>db

0 0 0 4 - IRTOD
        ir>ird

0 0 1 5 - FIX1
        chr>irb                 if irc needs to be replaced,
                                do access and transfer chr
                                to irb, else no activity.
        ! force miss regardless of whether odd or even
        change of flow
        fetch instr

1 0 0 6 - 2TOC
        irc2>irc
        irc>ir
        use pipe

0 0 0 8 - CLRA
        clear irc2[14]
        ira>ab                          zxtd 8 -> 32

0 0 0 9 - STIRA
        db>>ira
        ira>pb>irc2

0 0 0 11 - ATOC
        db>>ira
        ira>pb>irc

0 0 1 13 - EUD
        chr>irb
        irb>pb>imh,iml
        fetch instr

1 0 0 14 - CTOD
        irc>ir,ird
        irb>irc
        use pipe

1 0 1 15 - TUD
```

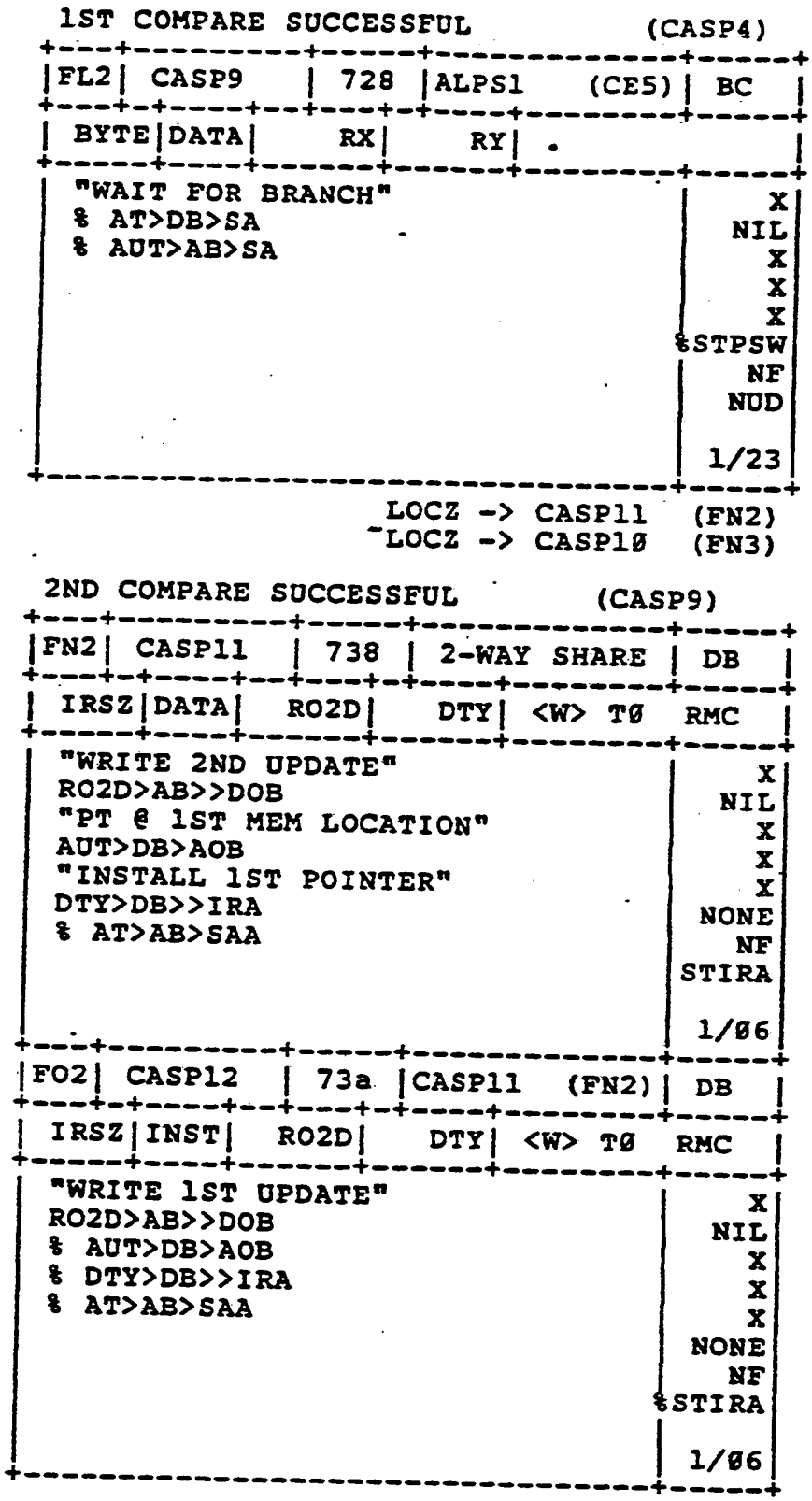
```
        chr>irb
        irb>pb>imh,iml,irc
        irc>ir
        use pipe
        fetch instr

Ø 1 1 15 - TOAD
        chr>irb
        irb>pb>imh,iml,irc
        irc>ir
        change of flow
        fetch instr
```

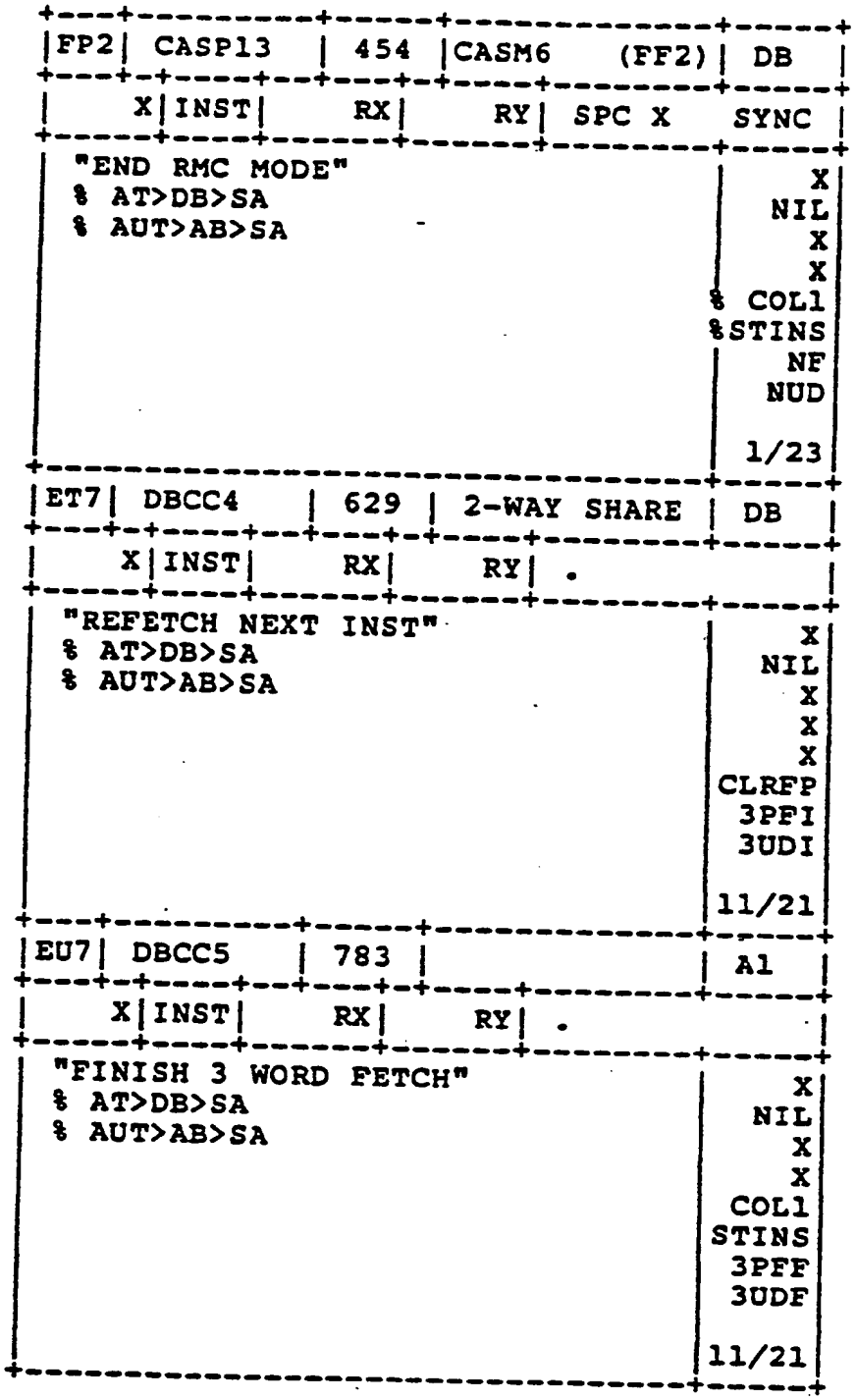
APPENDIX II
MICROINSTRUCTION LISTING

```
PAPA CAS
CAS2 DW1:DW2,DO1:DO2,(RZ1):(RZ2)
A1
+---+---------+-----+-------------+-----+
|FG1| CASP1   | 367 |             | DB  |
+---+-+----+--+---+-+----+--------+-----+
|   X|INST|    DT|    RY| .              |
+----+----+------+------+--------+-------+
| "STORE 1ST POINTERS"                |   X|
| IM>DB>DT                            | NIL|
| "LATCH RMC ADDRESS INTO A4"         |   X|
| % AT>DB>SAA                         |   X|
| % AUT>AB>SA                         |   X|
|                                     |    |
|                                     |  NF|
|                                     |UPIPE|
|                                     |    |
|                                     |1/26|
+---+---------+-----+-------------+-----+
|FH1| CASP2   | 654 |             | DB  |
+---+-+----+--+---+-+----+--------+-----+
| IRSZ|DATA|   RZ2|   RW2| <>  T1  | RMC |
+-----+----+------+------+--------+-----+
| "READ 1ST DEST"                     |   X|
| RZ2A>DB>>AOB                        | NIL|
| RZ2A>DB>AUT                         |   X|
| "STORE 1ST COMPARE VALUE"           |   X|
| RW2D>AB>>REGB                       |   X|
| "2ND POINTER TO IRC2"               |NONE|
| IN>DB>>IRA                          |  NF|
| "FORCE SYNC"                        |USTIR|
| RW2D>AB>>DOB                        |    |
| % AUT>AB>SAA                        |2/17|
+---+---------+-----+-------------+-----+
|FI1| CASP3   | 368 |             | DB  |
+---+-+----+--+---+-+----+--------+-----+
| IRSZ|INST|   RZ2|    RY| <>  T1  | RMC |
+-----+----+------+------+--------+-----+
| "READ 2ND DEST"                     |   X|
| RZ2A>AB>>AOB                        |COL2|
| "DO 1ST COMPARE"                    | CC2|
| DBIN>DB>ALU>ALUT                    |   X|
| REGB>AB>ALU                         |   X|
| "STORE 1ST DEST"                    |NONE|
| DBIN>DB>AT                          |  NF|
|                                     | NUD|
|                                     |    |
|                                     |1/21|
+-------------------------------------+----+
```









2ND COMPARE UNSUCCESSFUL (CASP9)

FN3	CASP1Ø	739	PSPF2 (GJ8)	DB
X	INST	RX	RY	SPC X	SYNC
"END RMC MODE"				%	ADD
% REGB>DB>AU>AOB					NIL
% 18>AU					X
% AUT>AB>SA					X
					X
					NONE
					NF
					NUD
					1/23

CASP6 (FM1)

RTE DEST FOR PAPA CAS

FH2	CASPR	455	CASM1 (FA1)	DB
X	INST	DT	RY	.	
"INITIALIZE IRC2 (FOR RTE)"					X
DT>DB>>IRA				%	AND
"LATCH RMC ADDRESS INTO A4"					X
% REGB>AB>>REGA					X
% Ø>ALU>ALUT					X
% AT>DB>SAA					STCRC
% AUT>AB>SAA					NF
					STIRA
					1/23

CASP2 (FH1)

Claims

1. In a data processor, a method for changing a value at a first location and a value at a second location, the method comprising the steps of:

providing as a first compare value the value at the first location;

providing as a second compare value the value at the second location;

providing a selected first swap value;

providing a selected second swap value; and then

performing in an uninterruptable sequence the steps of:

comparing the first compare value to the value at the first location;

comparing the second compare value to the value at the second location; and

if both of the first and second compare values are the same as the respective values at the first and second location;

storing the first swap value at the first location; and

storing the second swap value at the second location.

2. In a data processor, a method for changing the contents of a linked list which requires changing contents at two pointer locations in order to effect a change in said list, the method comprising the steps of:

providing as a first compare value the value at a first pointer location;

providing as a second compare value the value at a second pointer location;

providing a selected first swap value;

providing a selected second swap value; and
performing in an uninterruptable sequence the steps of:
comparing the first compare value to the value at said first pointer location;
comparing the second compare value to the value at said second pointer location; and
if both of the first and second compare values are the same as the respective values at the first and second pointer locations;
storing the first swap value as the value at the first pointer location; and
storing the second swap value as the value at the second pointer location.

3. In a data processor having a linked list which has an access counter associated therewith wherein contents at a pointer location and at the access counter are changed in order to effect a change in the list, a method for effecting a change in the list, comprising the steps of:
preparing for the excution of an instruction to be performed by the data processor by:
storing a first compare value, said first compare value read at a said pointer location which must be altered to effect said change in the list;
storing a second compare value, said second compare value read at the access counter;
storing a first swap value, said first swap value to be swapped for the value present at the pointer location in order to effect said change in the list; and
storing a second swap value, said second swap value to be inserted into the access counter; and

executing the instruction, wherein the instruction writes the first swap value into the pointer location and writes the second swap value into the access counter if, after the instruction commences, the value present at the pointer location is the same as the first compare value and the value present at the access counter is the same as the second compare value.

4. In a data processor, means for changing the contents of a linked list which requires changing contents of first and second pointer locations, comprising:

means for providing as a first compare value the value at said first pointer location;

means for providing as a second compare value the value at said second pointer location;

means for providing a first swap value;

means for providing a second swap value; and

means for performing in an uninterruptable sequence the steps of:

comparing the first compare value to the value at said first pointer location;

comparing the second compare value to the value at said second pointer location; and

if both of the first and second compare values are the same as the respective values at the first and second pointer location then:

storing the first swap value as the value at first pointer location; and

storing the second swap value as the value at the second pointer location.

5. In a data processor having a linked list which has an access counter associated therewith wherein contents at a pointer location and at the access counter are

changed in order to effect a change in the list, means for effecting a change in the list, comprising:

means for preparing for the excution of an instruction to be performed by the data processor in which said means for preparing:

stores a first compare value, said first compare value read at said pointer location which must be altered to effect said change in the list;

stores a second compare value, said second compare value read at the access counter;

stores a first swap value, said swap value to be swapped for the value present at said pointer location in order to effect said change in the list; and

stores a second swap value, said second swap value to be inserted into the access counter to be incremented; and

means for executing the instruction, wherein the instruction writes the first swap value into pointer location and writes the second swap value to the access counter if, after the instruction commences, the value present at the first pointer location is the same as the first compare value and the value present at the access counter is the same as the second compare value.

6. In a data processor, means for changing the contents of a value at a first location and a value at a second location, comprising:

means for providing as a first compare value the value at said first location;

means for providing as a second compare value the value at said second location;

means for providing a first swap value;

means for providing a second swap value; and

means for performing in an uninterruptable sequence the steps of:

comparing the first compare value to the value at said first location;

comparing the second compare value to the value at said second location; and

if both of the first and second compare values are the same as the respective values at the first and second locations then:

storing the first swap value as the value at the first location; and

storing the second swap value as the value at the second location.

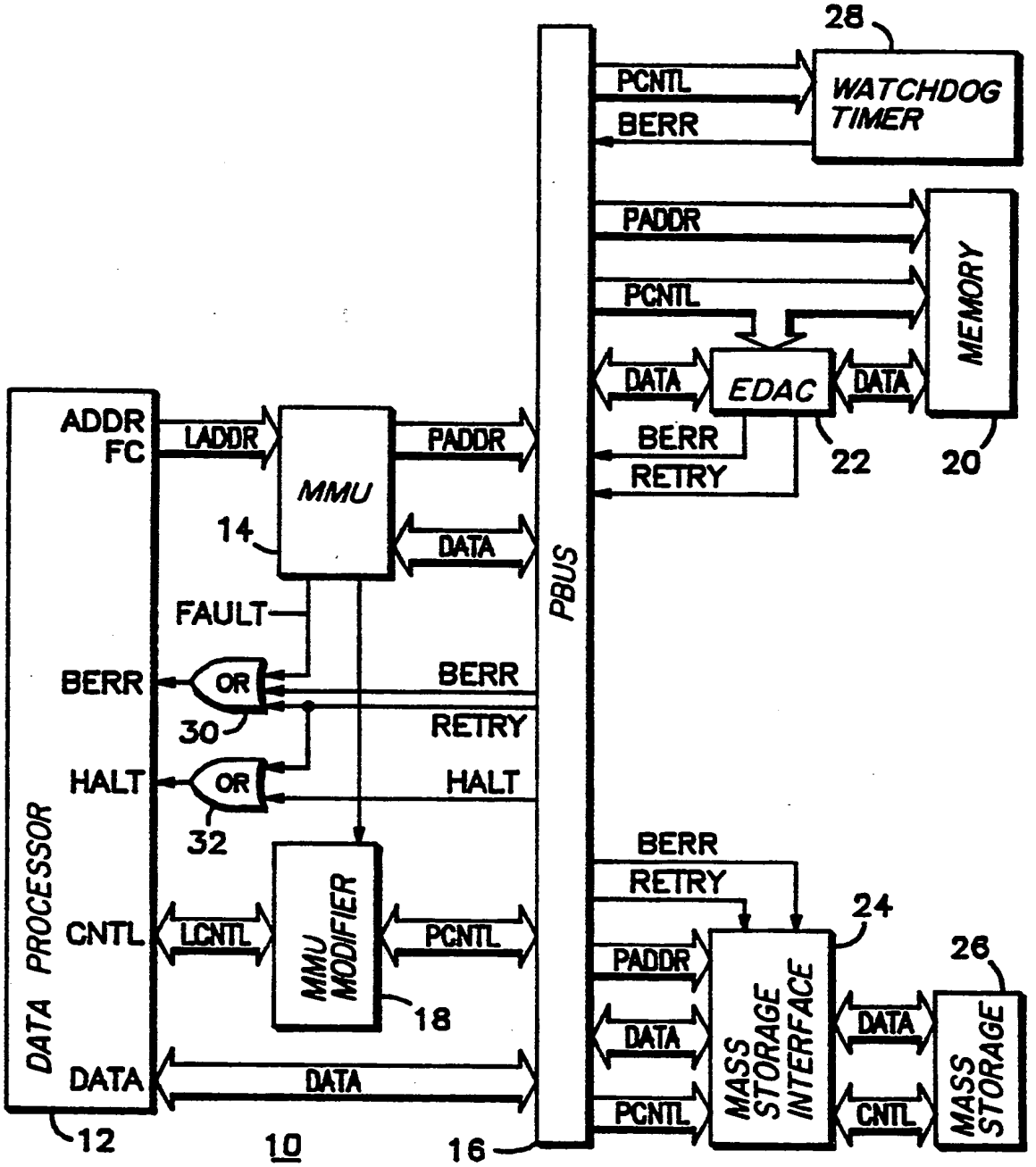


FIG. 1

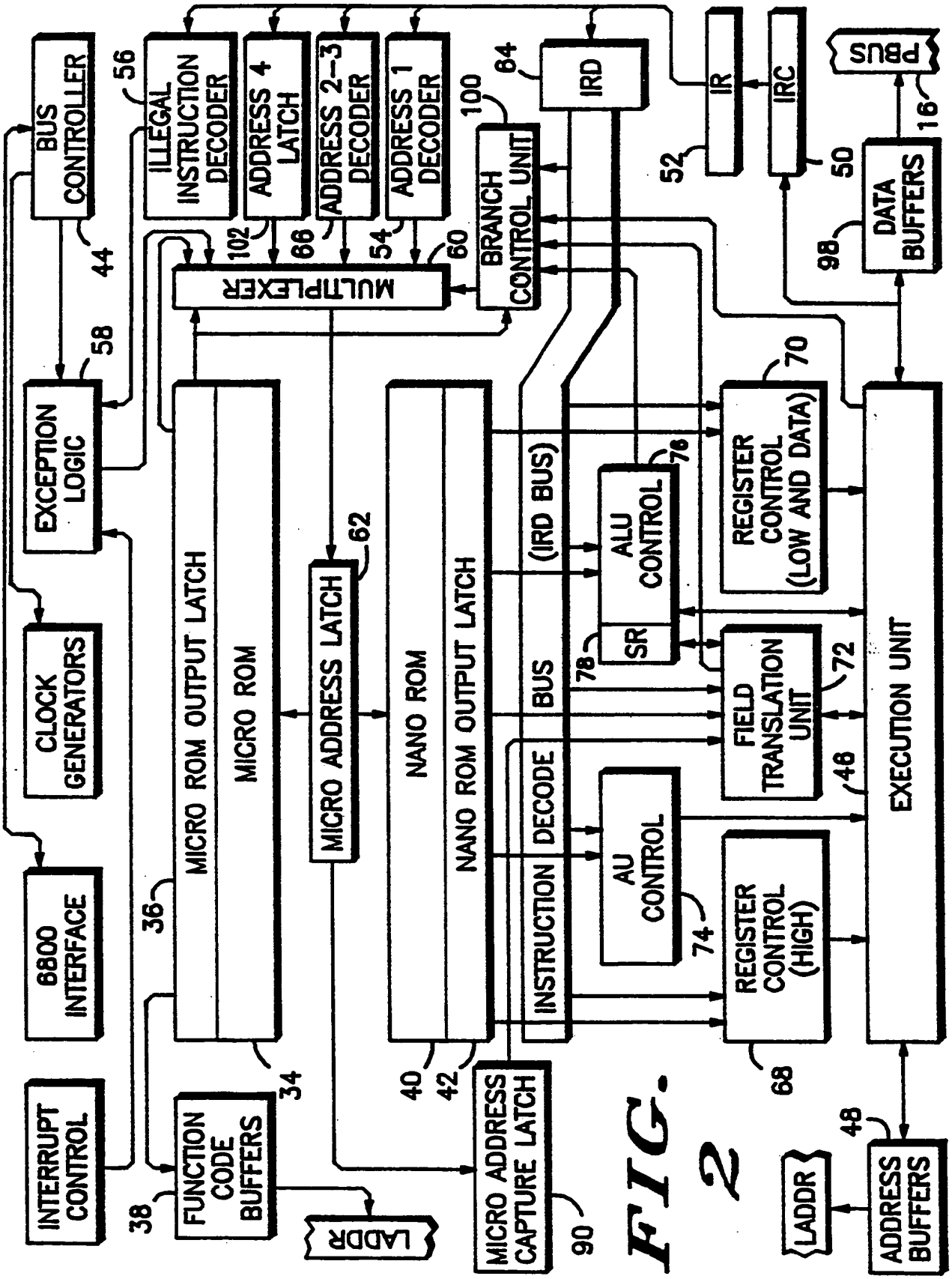


FIG. 2

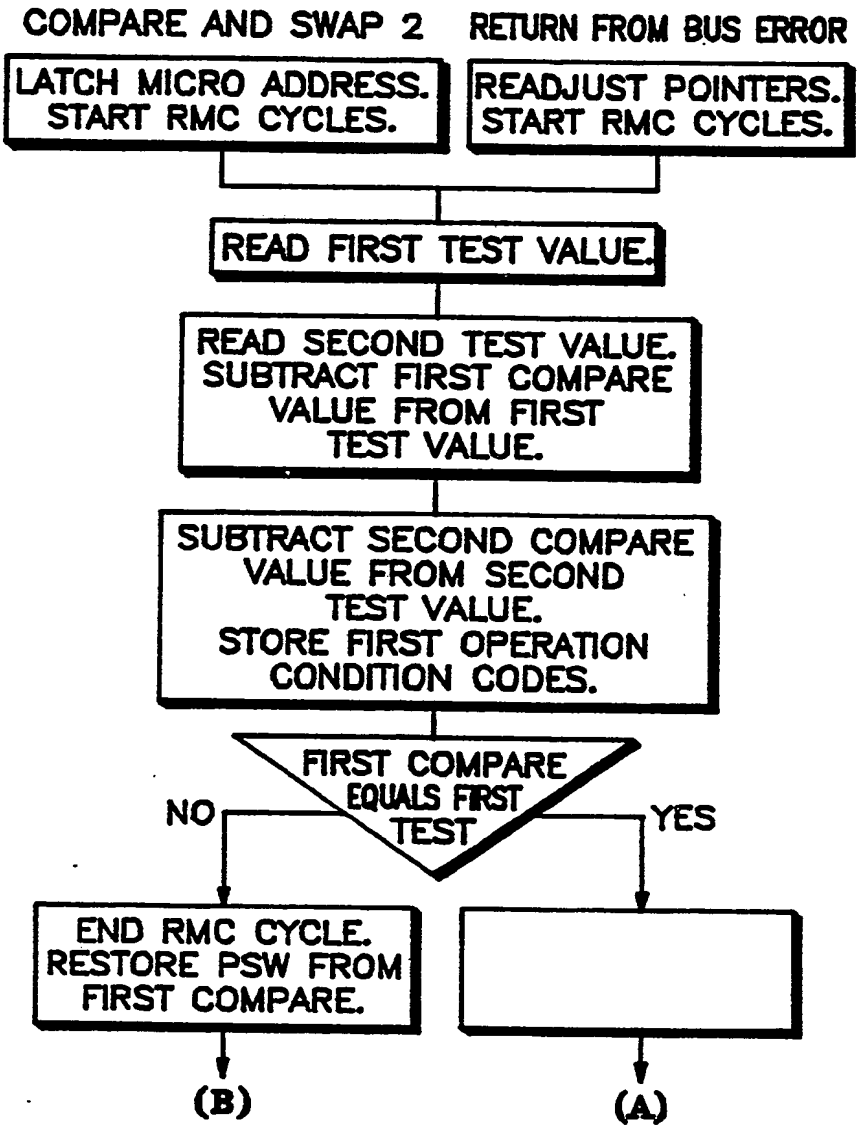


FIG. 3A

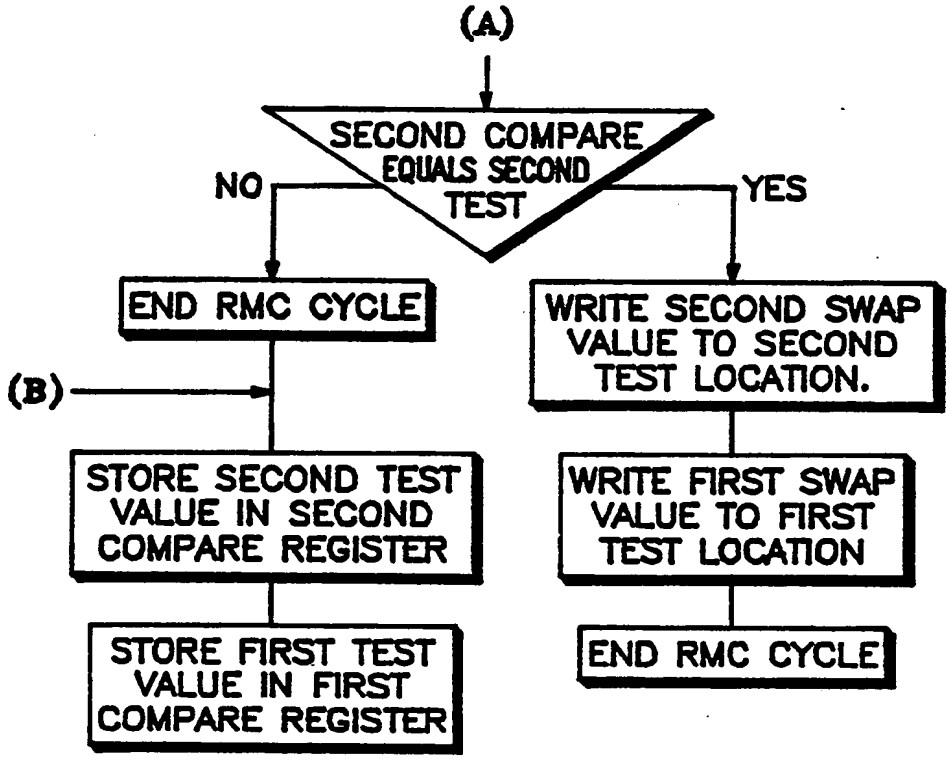


FIG. 3B

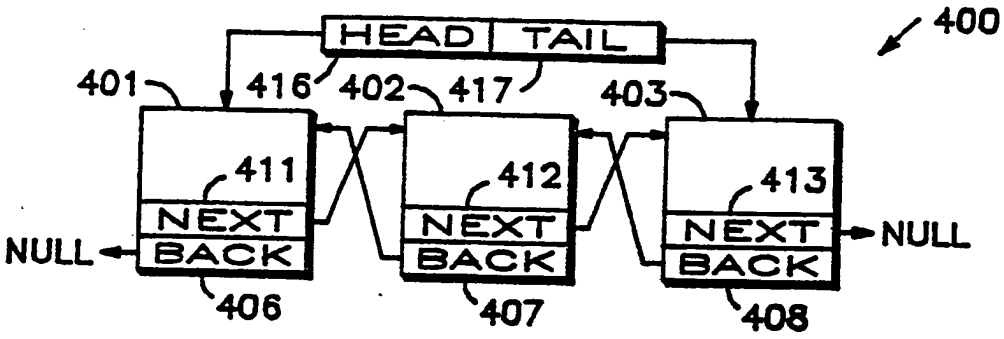


FIG. 4A

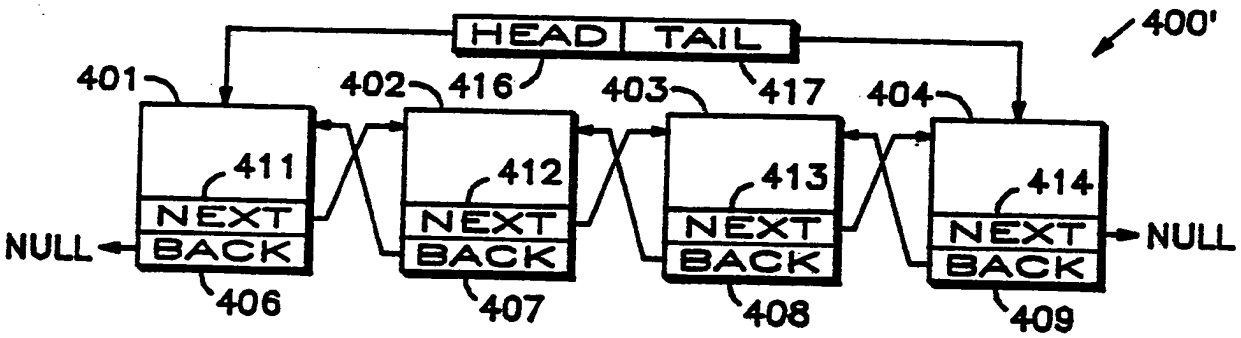


FIG. 4B

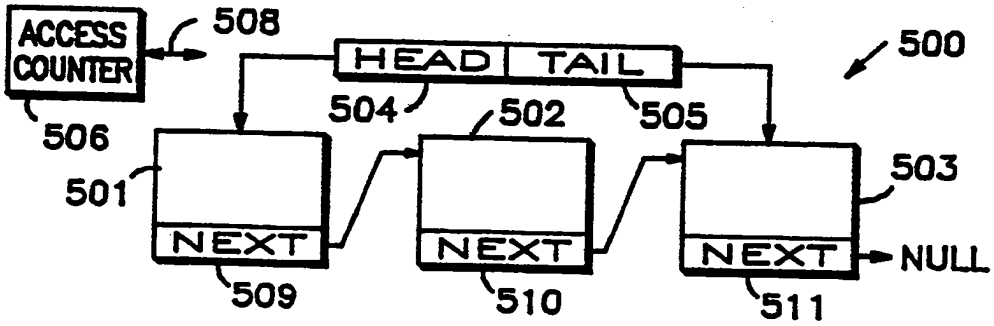


FIG. 5A

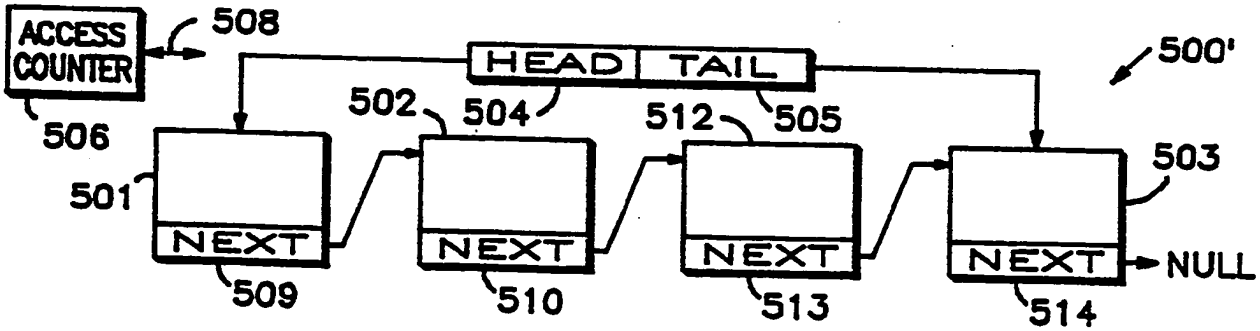


FIG. 5B

INTERNATIONAL SEARCH REPORT

International Application No PCT/US85/00670

I. CLASSIFICATION OF SUBJECT MATTER (if several classification symbols apply, indicate all) [3]

According to International Patent Classification (IPC) or to both National Classification and IPC

INT. CL.[4] G06F 7/00, 9/00, 15/00

II. FIELDS SEARCHED

Minimum Documentation Searched [4]

Classification System	Classification Symbols
U.S.	364/200, 300, 900

Documentation Searched other than Minimum Documentation to the Extent that such Documents are Included in the Fields Searched [5]

III. DOCUMENTS CONSIDERED TO BE RELEVANT [14]

Category *	Citation of Document, [16] with indication, where appropriate, of the relevant passages [17]	Relevant to Claim No. [18]
Y,P	US, A, 4,497,023, (Moorer), 29 January 1985	1-6
A	US, A, 3,972,026, (Waltman), 27 July 1976	1-6
A	US, A, 4,429,360, (Hoffman et al.), 31 January 1984	1-6
A	US, A, 4,281,393, (Gitelman et al.), 28 July 1981	1-6
A	US, A, 4,041,462, (Davis et al.), 09 August 1977	1-6

* Special categories of cited documents: [15]

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

IV. CERTIFICATION

Date of the Actual Completion of the International Search [2]	Date of Mailing of this International Search Report [2]
09 May 1985	05 JUN 1985
International Searching Authority [1]	**Signature of Authorized Officer** [20]
ISA/US	William A. Nelson

Form PCT/ISA/210 (second sheet) (October 1981)